Unittesting in Java

Unittesting is the most granular level of testing in software development, focusing on testing individual functions of code, in isolation. The main goal is to verify that each unit behaves as expected in a controlled environment, typically without relying on external systems like databases or APIs.

Junit

JUnit is a very common testing framework to write unittests in Java. Usually, developers write their own unittests as part of the development process.

Installing the Junit dependency artifacts

Important

During the tutorial, you'll be working on your fork of the SpotifyCatalogAPI presented in the previous tutorial[1].

Make sure the following dependency is listed in your pom.xml file:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.9.0</version>
  <scope>test</scope>
</dependency>
```

Keep in mind that the <scope> of the dependency is defined as test because it indicates that the dependency is only required during the test phase of the build lifecycle. This means that the junit-jupiter-engine library will be used exclusively for testing purposes, and it will not be included in the final artifact.

Running unittests as part of the build lifecycle

In Maven, unit tests are integrated into the build lifecycle and are automatically executed during the test phase.

The Maven maven-surefire-plugin is the tool responsible for running the unittests during this phase:

```
$ mvn help:describe -Dcmd=test
...

[INFO] 'test' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
```

...

Under the src/test/java/com/example/catalog dir in the SpotifyCatalogAPI repo, you are already provided with a unittest named HelloControllerTest.

This test is designed to verify that the HelloController controller class behaves as expected.

1. **Run the mvn test command to see the unittests execution in action. Upon successful execution, the unit tests should pass, and you'll see an output similar to:**
2. **...**
3.
4. **[INFO] BUILD SUCCESS**
5. **Try to break the test by changing the code of the HelloController controller (under src/main/java/com/example/catalog/controller/HelloController.java). For example, modify the response text returned by the home() method.**

Note

Unittest files are in the form *Test.java and located in the src/test/java directory. These files are automatically discovered and executed during the Maven build process.

Writing Unittests

Here is a simple unittest:

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;


class MathTest {


  @Test

  void addition() {

    int actual = 1 + 1;

    int expected = 2;


    assertEquals(expected, actual);

  }
```

```
  @Test
  void division() {
    float actual = 6 / 3.0;
    int expected = 2;

    assertEquals(expected, actual);
  }

}
```

We say that MathTest is a test class which organizes the addition and the division test cases.

A Test Case is a function annotated by @Test and verifies a specific functionality of the code.

The addition test case, for example, simply checks that 1 + 1 = 2.

The assertEquals assertion is used to verify that the expected value matches the actual value in a test case. If the assertion fails, the test case is reported as a failure.

The @Test syntax is part of something called annotations in the Java language.

Annotations

Annotations are kind of labels (a.k.a. metadata), provide information about the program, that is not part of the program itself. Annotations, in the form of @AnnotationName, can be applied to classes, methods, function parameters and more. They are used to provide additional information to the compiler or during runtime environment.

In the above example, the @Test tells JUnit that "this function is a test case, if the assertEquals assertion inside the addition method fails, report this testcase as a failure" If we won't annotate the method with @Test, Junit will not recognize it as a test case.

Junit has a few more useful annotations.

Exercises

🖍️ Unittests for the SpotifyUtils class

Tip

Recap your regex knowledge if needed!

1. **First, review the Spotify API docs on URIs and IDs to become familiar with these key identifiers. You've probably notices that we use these URIs and IDs in our SpotifyCatalogAPI service.**
2. **Review the src/main/java/com/example/catalog/utils/SpotifyUtils.java class. It provides utility methods to validate Spotify URIs and IDs, and to interact with the real Spotify API.**
3. **Under src/test/java/com/example/catalog/SpotifyUtilsTest.java you'll find some partial test cases. Follow the below instructions to complete:**
    - **Remove the @Disabled ann to execute this test class as part of your Maven build lifecycle.**
    - **You are already given a complete test cases for the isValidId method. Your goal is to implement the method itself within the SpotifyUtils class, and to make sure your implementation is passing the tests. This approach of writing tests before the code implementation is known as Test-Driven Development (TDD).**
    - **Write test cases for isValidURI.**
    - **Write test cases for getSpotifyClient (since this feature is not implemented yet, you only have to assert that exception is thrown upon an invalid arguments input).**
4. **When done, after each one of the SpotifyUtils's method is covered by a test cases, run the mvn test command and make sure the test cases are executed without failures.**

🖍️ Unittests for the CatalogUtils class

The CatalogUtils class under src/main/java/com/example/catalog/utils/CatalogUtils.java contains some utilities methods to manipulate songs data.

Use the below skeleton to create a test class with test case(s) for each util method.

```
package com.example.catalog;

import com.example.catalog.utils.CatalogUtils;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.List;

class CatalogUtilsTest {

  private CatalogUtils catalogUtils;
  private List<JsonNode> songs;
  private ObjectMapper objectMapper;

  @BeforeEach
  void setUp() throws Exception {
    catalogUtils = new CatalogUtils();
    objectMapper = new ObjectMapper();

    // Sample song data for testing. TODO - Add more songs
    String jsonData = """
          [
            {
              "duration_ms": 200040,
              "name": "Blinding Lights",
              "popularity": 87,
              "album": {
```

```
            "name": "After Hours",
            "release_date": "2020-03-20",
            "total_tracks": 14
          },
          "artists": [
           {
            "name": "The Weeknd"
           }
          ]
         }
       ]
      """;
    songs = new ArrayList<>();
    objectMapper.readTree(jsonData).forEach(songs::add);
  }
}
```

- You have to add more songs to jsonData (take a look in src/main/resources/data/popular_songs.json to get some examples) to ensure it includes a variety of cases, such as different artists, release years, durations, to comprehensively cover all scenarios.
- Make sure you understand the @BeforeEach annotation's behaviour.

🖍️ Nested testing for the LRUCache class

In software development, caching is a technique used to store frequently accessed data temporarily, improving performance by reducing the need for repeated computations or costly I/O operations. There are various caching techniques, and one of the most common is Least Recently Used (LRU), which evicts the least recently accessed items when the cache reaches its maximum size.

The LRUCache class can be used to manage data caching in your service. Integrate the LRUCache in the getAlbumById controller. Frequent requests for the same album id should now be retrieved directly from the cache instead of reading the JSON file each time.

Test your app after the modifications.

We now write a test class for the LRUCache class.

In JUnit, nested test classes allow you to group related test cases, improving readability and organization. Read the Junit docs and familiarize yourself with nested test classes.

Then, complete the LRUCacheTest nested test class according to the following structure:

- **when instantiated with capacity 3:**
    - **cache is initially empty**
    - **throws NullPointerException when getting a null key**
    - **throws NullPointerException when setting a null key**
    - **after adding 2 elements:**
        - **cache contains the added elements**
        - **cache maintains the same size and content after adding duplicate keys**
        - **ablity to add one more element without revoke another existed keys**
        - **maintains correct order of elements after accessing an existing element**
        - **updates the value of an existing key when set again**
    - **after adding 3 elements:**
        - **size does not exceed capacity after adding more elements**
        - **evicts the least recently used element when the cache exceeds its capacity**
        - **when cleared:**
            - **cache is empty**
            - **added elements are not accesible**
    - **when working with boundary conditions:**

- **handles adding elements up to exactly the capacity without eviction**
- **evicts only the least recently used element when capacity is exceeded repeatedly**

Add more cases of your own (e.g. retrieve non-existing key). Execute your test class.

Note

Boundary conditions refer to the limits or edges of input values or scenarios that a system or function must handle, such as maximum, minimum, or null values. Testing these conditions ensures the system behaves correctly and robustly under extreme or unusual circumstances.

Footnotes