Tab 1

This document outlines the testing approach for the TMDB Discover demo website, focusing on the filtering and pagination functionalities.

**1. Objective**

The primary objective is to verify the correctness and robustness of the filtering and pagination features on the TMDB Discover platform.

**2. Scope**

**In-Scope:**

* Filtering by Categories: Popular, Trending, Newest, Top Rated
* Filtering by Title search
* Filtering by Type: Movies or TV Shows
* Filtering by Year
* Filtering by Rating
* Filtering by Genre selection
* Pagination functionality (Next and Previous buttons, Direct selection of Page by clicking on page number)

**Out-of-Scope:**

* The accuracy of the movie/TV show data
* Layout and responsive design testing
* Performance and load testing
* Standalone API-level testing

**3. Automation Approach**

For this project, Cucumber BDD, Selenium WebDriver, @Slf4j Logging, JUnit Assertions, and HTML Reports would be suitable. This approach combines powerful tools to create robust, readable, and maintainable automated web tests.

**4. Test Case Identification**

Test Cases identified are based on website functionality that allows users to find movies and TV shows based on various criteria.

PS: Test Cases are elaborated in .xls file ‘RR-TMBD-TestCases-And-Defect Details

’

**Functional Test Cases:**

* Basic functionality - To check the Movies/TV Shows section and verify that a default list of trending/popular content is displayed properly as this is the core functionality and the first impression for the user, it ensures the basic data retrieval and display are working.
* Search Functionality - Allows users for more granular searching.
* Filtering - Filtering is central to the user experience. It allows users to narrow down options and find specific content, These test cases ensure the logic for combining and applying filters like Genre, rating, Year Of Release
* Pagination - Essential for handling large sets of data/content and providing a good user experience when browsing through results.
* Edge Cases for Filters - Ensures the system handles unexpected or extreme inputs gracefully, preventing errors and improving robustness.
* Usability Test Cases - A well-designed UI is crucial for user adoption and satisfaction.

**5. Approach and Details for Automation Suite**

For now considered only automation of ‘**Smoke Test Automation Suite’**

a. Project Setup done using Maven where pom.xml , will be the central point for managing dependencies.

Below are few important ones -

* Cucumber-Java: For writing BDD features and step definitions.
* Cucumber-JUnit: To run Cucumber tests with JUnit.
* Selenium Java: For browser automation.
* JUnit: For assertions within your step definitions.
* Lombok: To use @Slf4j for concise logger creation.
* Cucumber Reporting: For generating comprehensive HTML reports.

**b. Feature Files (.feature)**

Written in Gherkin syntax (Given-When-Then). It describes the desired behavior from a business perspective. It uses keywords like Feature, Scenario, Given, When, Then, And, But, Background and Scenario Outline (or Scenario Template) to structure the descriptions. Feature: keyword at the top of the file provides a high-level description of the functionality being discussed. Within a feature, individual **scenarios** (defined by Scenario: ) illustrate specific examples of how the feature should behave. Each scenario describes a concrete instance of a user interaction or system behavior, following the "Given-When-Then" structure:

* **Given:** Describes the initial context or preconditions.
* **When:** Describes the action or event that triggers the behavior.
* **Then:** Describes the expected outcome or result.
* **And / But:** Used to extend Given, When, or Then statements for readability.

**c. Step Definitions (.java)**

Bridge the gap between feature files and automation code, all code logic is included in this file with annotated methods (@Given, @When, @Then) corresponding to Gherkin steps in the feature file.This is where Selenium WebDriver interacts with the web elements.

@Slf4j is used for Logging, it is added to step definition classes to easily log events, status, and debug information.

**d. Runner Class (.java)**

A JUnit test class that tells Cucumber where to find features and step definitions.Also it has @CucumberOptions to configure the test run, the plugin option is used to specify output formats, including the HTML report or JSON report.

**e. Hook file (.java)**

In Cucumber BDD, hook file is a block of code that allows you to execute logic at specific points in the Cucumber test lifecycle, outside of your regular step definitions. They are extremely useful for managing setup and teardown activities that are common across multiple scenarios or steps.For example, @Before Hook Executes before each scenario(Launching a web browser), @After Hook Executes after each scenario, regardless of whether the scenario passed or failed(Closing the web browser)

**Summary of the Approach:**

* BDD with Cucumber : Test scenarios are written in a human-readable format.
* Selenium WebDriver : Automate browser interactions in step definitions.
* @Slf4j Logging : Integrate effective logging within step definitions to trace execution flow, debug issues, and provide useful runtime information. This helps immensely in understanding what the test is doing and where it might be failing.
* JUnit Assertions : Use JUnit's powerful assertion methods (assertTrue, assertEquals, etc.) to validate expected outcomes within your @Then steps.
* HTML Reporting : Generate clear, comprehensive, and shareable HTML reports (especially with maven-cucumber-reporting) that provide an overview of test execution, detailed scenario results, and step-by-step breakdowns, making it easy to analyze failures and share results with stakeholders.