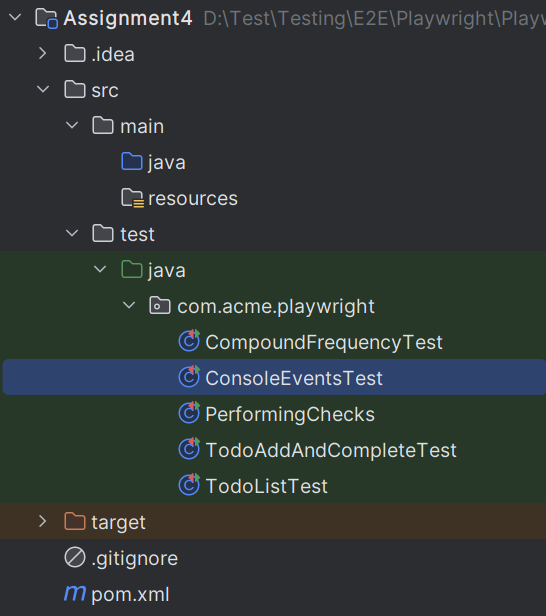
# Playwright for Java - Hands-On Assignment 4

# Project Structure



# Problem Statement 1: Automating a React-based TodoMVC App with Playwright Fixtures

## Objective:

Practice using Playwright’s JUnit 5 @UsePlaywright fixtures to test a modern JavaScript web app (React TodoMVC), ensuring core to-do list functionalities work as expected in a browser run in non-headless mode.

## Steps:

1. Use the @UsePlaywright annotation to enable Playwright’s built-in JUnit 5 fixture support for automatic Page injection.
2. Write the following test cases in the same class:

### Test 1: Verify Empty List on Load

a. Navigate to https://todomvc.com/examples/react/dist/

b. Assert that the .todo-list li elements count is 0.

### Test 2: Add a New Todo

a. Navigate to the TodoMVC app.

b. Fill the .new-todo input with "Learn Playwright" and press Enter.

c. Assert that the todo list contains exactly 1 item.

d. Assert that the first todo item contains the text "Learn Playwright".

### Test 3: Mark a Todo as Completed

a. Add a todo "Complete me".

b. Click the first .toggle checkbox.

c. Assert that the corresponding <li> element has a CSS class containing "completed".

### Test 4: Clear Completed Todos

a. Add a todo "To be cleared".

b. Mark it as completed and click the "Clear completed" button.

c. Assert that the todo list is now empty.

### Test 5: Filter Active and Completed Todos

a. Add two todos: "Active task" and "Completed task".

b. Mark the second task as completed.

c. Click the "Active" footer link and assert only "Active task" is shown.

d. Click the "Completed" footer link and assert only "Completed task" is shown.

## Expected Output:

* Test 1: Passes if the todo list is empty on initial page load.
* Test 2: Passes if the new todo appears correctly in the list.
* Test 3: Passes if the completed todo has the "completed" CSS class.
* Test 4: Passes if clearing completed todos removes them from the list.
* Test 5: Passes if the filters correctly show only active or completed todos.

# Problem Statement 2: Automating a React TodoMVC App with Playwright Fixtures

## Objective:

Practice automating a basic "Add and Complete Todo" flow in the React-based TodoMVC app using Playwright’s JUnit 5 fixtures, ensuring that newly added items appear correctly and can be marked as completed.

## Steps:

1. Use the @UsePlaywright annotation to enable Playwright’s built-in JUnit 5 fixture support for automatic Page injection.

1. Write a test method to:

**Test: Verify Adding and Completing a Todo**

a. Navigate to the React TodoMVC app at https://todomvc.com/examples/react/dist/.

b. Fill the new todo input (.new-todo) with the task "Buy milk" and press Enter.

c. Verify the newly added todo item appears in the list.

d. Mark the todo as complete by checking the toggle box (.toggle).

e. Assert that the completed todo has the CSS class "completed".

**Expected Output:**

Test passes if:

* The new todo "Buy milk" appears in the list.
* The todo’s HTML element has the "completed" class after being marked complete.

# Problem Statement 3: Automating a Textarea Submission and Verification with Playwright Fixtures

## Objective:

Practice using Playwright’s JUnit 5 @UsePlaywright fixtures to test textarea input on a live demo form page, ensuring the entered text is captured correctly after form submission.

## Steps:

1. Use the @UsePlaywright annotation to enable Playwright’s built-in JUnit 5 fixture support for automatic Page injection.
2. Navigate to the given sample page URL: https://deformdemo.pylonsproject.org/textarea/
3. Locate the <textarea> element using a CSS selector.
4. Assert that the textarea is visible and within the viewport.
5. Assert that the textarea is initially empty.
6. Fill the textarea with the message "Hello Demo".
7. Assert that the textarea value matches "Hello Demo".
8. Click the submit button.
9. Locate the <pre> block with id="captured" showing the submitted data.
10. Assert that this captured block contains the entered message "Hello Demo".

## Expected Output:

* The textarea is empty initially.
* The textarea value updates correctly after filling.
* The captured output on the page contains "Hello Demo" after submission.

# Problem Statement 4: Compound Frequency and Future Value Verification

**Objective:** Automate a test for the Compound Interest Calculator on CalculatorSoup using Playwright + JUnit 5. Verify that selecting different compounding frequencies updates the future value correctly and that more frequent compounding results in a higher FV.

**Steps:**

1. Navigate to the CalculatorSoup Compound Interest Calculator page: <https://www.calculatorsoup.com/calculators/financial/compound-interest-calculator.php>.
2. Fill in the following common inputs:
   * Principal = 1000
   * Annual interest rate = 5%
   * Time in years = 10
3. Click the **Calculate** button to trigger the calculation.
4. Verify FV for different compounding frequencies:
   * **Annually (n = 1)**: read FV and assert it matches the formula A = P\*(1 + r/n)^(n\*t) within tolerance.
   * **Monthly (n = 12)**: read FV and assert it matches the formula within tolerance.
5. Assert that **Monthly FV > Annual FV**.

**Stubbed Code Template:**

package com.acme.playwright;  
  
import com.microsoft.playwright.Page;  
import com.microsoft.playwright.junit.UsePlaywright;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
@UsePlaywright  
public class CompoundFrequencyTest {  
  
 private static final String URL =  
 "https://www.calculatorsoup.com/calculators/financial/compound-interest-calculator.php";  
  
 @Test  
 void selectingCompoundFrequencyUpdatesResult(Page page) {  
 // Navigate to the calculator page  
 page.navigate(URL);  
  
 // --- Fill common inputs ---  
 setNumber(page, "#P", "1000"); // Principal  
 setNumber(page, "#R", "5"); // Rate %  
 setNumber(page, "#t", "10"); // Years  
  
 // --- Trigger initial calculation ---  
 clickCalculate(page);  
 waitUntilAnswerUpdates(page);  
  
 // --- Annual compounding ---  
 selectCompoundByValue(page, 1); // n = 1 for annually  
 clickCalculate(page);  
 waitUntilAnswerUpdates(page);  
 double annualFV = readAnswerA(page);  
 double annualExpected = futureValue(1000, 0.05, 1, 10);  
 assertWithin(annualFV, annualExpected, 1.0, "Annual FV should match formula");  
  
 // --- Monthly compounding ---  
 selectCompoundByValue(page, 12); // n = 12 for monthly  
 clickCalculate(page);  
 waitUntilAnswerUpdates(page);  
 double monthlyFV = readAnswerA(page);  
 double monthlyExpected = futureValue(1000, 0.05, 12, 10);  
 assertWithin(monthlyFV, monthlyExpected, 1.0, "Monthly FV should match formula");  
  
 // --- Monthly should be greater than Annual ---  
 assertTrue(monthlyFV > annualFV, "Monthly FV must be greater than Annual FV");  
 }  
  
 // ---------------- Helper Methods ----------------  
  
 /\*\* Fill a numeric input and trigger JS events \*/  
 private static void setNumber(Page page, String selector, String value) {  
 // TODO: implement fill and type events  
 }  
  
 /\*\* Select compounding frequency by value (1=Annual, 12=Monthly) \*/  
 private static void selectCompoundByValue(Page page, int nPerYear) {  
 // TODO: implement dropdown selection  
 }  
  
 /\*\* Click the 'Calculate' button \*/  
 private static void clickCalculate(Page page) {  
 // TODO: implement click  
 }  
  
 /\*\* Wait until the answer panel updates with the latest FV \*/  
 private static void waitUntilAnswerUpdates(Page page) {  
 // TODO: implement wait function  
 }  
  
 /\*\* Parse the Future Value from the answer panel \*/  
 private static double readAnswerA(Page page) {  
 // TODO: extract "A = $..." from #answer using regex  
 return 0;  
 }  
  
 /\*\* Calculate expected future value using formula: A = P\*(1 + r/n)^(n\*t) \*/  
 private static double futureValue(double principal, double rateAnnual, int nPerYear, double years) {  
 return principal \* Math.pow(1 + rateAnnual / nPerYear, nPerYear \* years);  
 }  
  
 /\*\* Assert actual and expected values match within a tolerance \*/  
 private static void assertWithin(double actual, double expected, double tol, String msg) {  
 // TODO: assert |actual - expected| <= tol  
 }  
}

**Explanatory Notes:**

* setNumber(...): fills numeric inputs and triggers input/change events to update the page.
* selectCompoundByValue(...): selects the desired compounding frequency from the dropdown.
* clickCalculate(...): clicks the Calculate button to trigger recalculation.
* waitUntilAnswerUpdates(...): waits until the FV panel is updated, ensuring the test reads the correct value.
* readAnswerA(...): extracts the Future Value from the text content of #answer.
* futureValue(...): computes the expected FV using the standard compound interest formula.
* assertWithin(...): allows numeric comparison within a small tolerance to account for rounding.

# Problem Statement 5: Capturing Console Messages and Page Errors

**Objective:** Use Playwright + JUnit 5 to monitor browser console messages and uncaught page errors, and assert that no errors occur during page load or basic interactions.

**Steps:**

1. Navigate to a publicly accessible page (e.g., https://example.com).
2. Capture all console messages:
   * Print type and text to console.
   * Collect messages of type error for assertion.
3. Capture uncaught page errors using onPageError:
   * Print exception messages.
   * Fail the test immediately if an error occurs.
4. Interact with existing clickable elements (e.g., first link <a>).
5. Assert:
   * No console errors occurred.
   * No uncaught page exceptions.

**Explanatory Notes:**

* onConsoleMessage(...): Captures all console messages (log, info, warn, error).
* onPageError(...): Captures uncaught JavaScript exceptions.
* Assertions verify no console errors or page exceptions occur.
* Interacting with existing elements ensures messages are captured from realistic user actions.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*