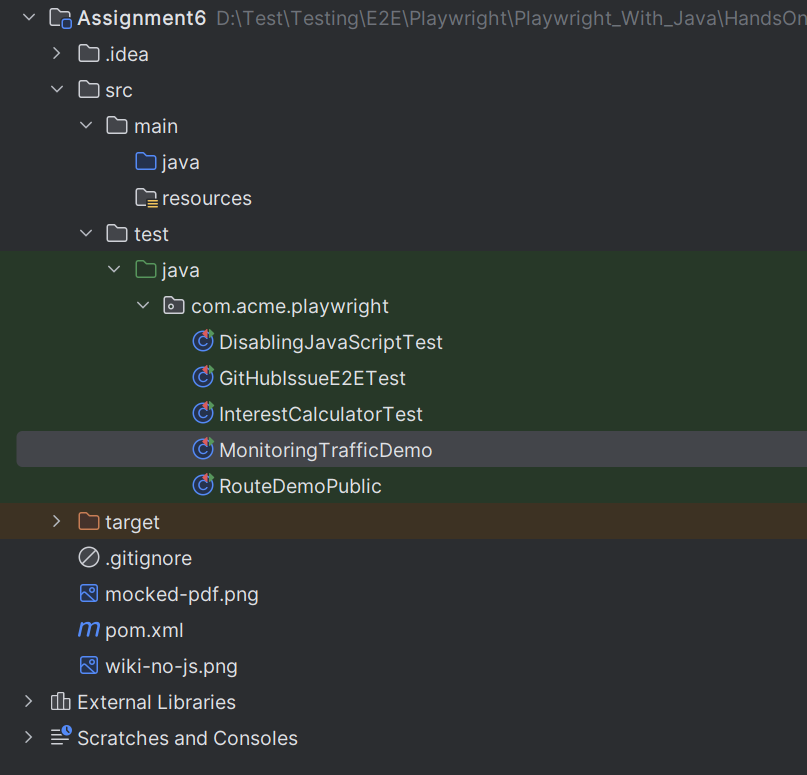
# Playwright for Java - Hands-On Assignment 6

# Project Structure



# Problem Statement 1: Validating a Simple Interest Calculator with Playwright

## Objective:

Practice using Playwright's locators and assertions to automate and validate the functionality of a simple interest calculator web page.

## Steps:

1. Use the @UsePlaywright annotation to enable fixture-based injection of a BrowserContext object.

Write a test method that:

1. Sets a default assertion timeout for the test using PlaywrightAssertions.setDefaultAssertionTimeout(2000).
2. Creates a new Page instance from the BrowserContext.
3. Navigates to the simple interest calculator URL: <https://www.calculator.net/simple-interest-calculator.html>.
4. Locates and fills the principal amount field with "1000", setting a specific timeout of 5000ms for this action.
5. Locates and fills the interest rate field with "5", also with a timeout of 5000ms.
6. Locates and fills the time period field with "2" (for two years), also with a timeout of 5000ms.
7. Clicks the "Calculate" button, setting a specific timeout for this action.
8. Locates the End Balance result from the table and asserts that its text content is "$1,100.00".
9. Locates the Total Interest result from the table and asserts that its text content is "$100.00".

## Expected Outcome:

* The test should pass successfully, confirming that the calculator correctly computes the simple interest and end balance based on the provided inputs.
* The console output should show a successful test run, indicating that all assertions passed. The test verifies that for a principal of $1,000 at a 5% rate over 2 years, the total interest is $100.00 and the end balance is $1,100.00.

# Problem Statement 2: Testing with Disabled JavaScript in Playwright

## Objective:

Practice simulating a user with JavaScript disabled in their browser context and asserting that a web application correctly displays a warning message.

**Steps:**

1. Use the @UsePlaywright annotation to enable fixture-based injection of a Playwright object.

Write a test method that:

1. Launches a new browser instance.
2. Creates a new BrowserContext and explicitly disables JavaScript.
3. Creates a new Page instance within this context.
4. Navigates the page to a public URL (e.g., <https://www.enable-javascript.com/>). This site is designed to show different content based on whether JavaScript is enabled.
5. Locates the warning message element that appears when JavaScript is disabled.
6. Asserts that the warning message is both visible and contains the correct text.

## Expected Outcome:

The test should pass successfully if:

* The test proves that Playwright can launch a browser with specific features disabled, such as JavaScript.
* The assertions confirm that the website's warning message for disabled JavaScript is correctly displayed, validating the application's behavior in a non-standard browser environment.

# Problem Statement 3: End-to-End Repo + Issue Workflow with Playwright API & UI

## Objective

Practice combining API testing and UI testing in Playwright by:  
- Creating a new GitHub repository via API.  
- Creating an issue in that repository via API.  
- Navigating to GitHub UI and validating that the issue is visible.  
- Cleaning up by deleting the repo after the test.

## Steps

1. Use the @UsePlaywright annotation to enable fixture-based injection of Playwright, Browser, and Page objects.  
  
2. In a @BeforeEach method:  
 - Initialize an APIRequestContext with GitHub API base URL.  
 - Provide authorization headers using a personal access token.  
 - Create a new repository via POST /user/repos.  
 - Assert that the response status is OK.  
  
3. In the @Test method:  
 - Use the API context to create a new issue in the repo (POST /repos/{username}/{repo}/issues).  
 - Extract the issue number from the JSON response.  
 - Open GitHub in the browser and navigate to the repo’s Issues tab.  
 - Locate the newly created issue using getByRole(AriaRole.LINK, …).  
 - Assert that the issue is visible and has the correct title.  
  
4. In the @AfterEach method:  
 - Delete the created repo via DELETE /repos/{username}/{repo}.  
 - Assert that the delete response status is 204 (No Content).

## Expected Outcome

Expected Outcome  
The test should pass if:  
- The repo is created successfully via API.  
- The issue is created successfully and retrievable from API response.  
- The issue appears correctly in the GitHub UI.  
- The repo is deleted after the test, leaving no residue.

# Playwright Assignment 4: Route Handling in Java

## Problem Statement

You are tasked with exploring network request interception in Playwright using page.route(). The assignment has two parts:  
1. Abort JavaScript requests on a site.  
2. Mock PDF responses with custom HTML.

## Part A: Abort JavaScript Requests

### Input

Test site: https://www.wikipedia.org/  
Task: Use Playwright to block all requests ending in .js.

### Steps

1. Intercept requests with page.route("\*\*/\*.js", route -> route.abort());  
2. Navigate to Wikipedia.  
3. Check if the search box (#searchInput) is visible.  
4. Take a screenshot.

### Expected Output

- The search input box is visible (HTML loads fine without JS).  
- Dynamic language switching does not work (because JS is blocked).  
- Screenshot saved as: wiki-no-js.png.

## Part B: Fulfill PDF Request with Mock HTML

### Input

Test site: https://httpbin.dev/  
Task: Intercept the /pdf request and return a mocked HTML response.

### Steps

1. Intercept PDF requests with page.route("\*\*/pdf", ...).  
2. Navigate to https://httpbin.dev/.  
3. Click the /pdf link.  
4. Fulfill the intercepted request with:  
 - Status: 404  
 - Content-Type: text/html  
 - Body: <h1>Not Found!</h1>  
5. Assert that the page body contains "Not Found!".  
6. Take a screenshot.

### Expected Output

- The PDF request is replaced with a mocked HTML page.  
- The page body contains: Not Found!  
- Screenshot saved as: mocked-pdf.png

# Playwright Assignment 5: Monitoring Network Traffic in Java

## Problem Statement

You are tasked with building Playwright tests that monitor and validate network traffic in real time. The assignment has three parts:  
  
1. Print all request/response activity.  
2. Collect and assert that all responses have valid status codes (200–299).  
3. Use the `.ok()` method to verify that no failing responses are present.

## Part A: Print Network Traffic

### Input

Test site: https://httpbin.dev/

### Task

Listen to requests and responses and print them to the console.

### Steps

1. Attach a `page.onRequest()` listener to print the request method and URL.  
2. Attach a `page.onResponse()` listener to print the response status code.  
3. Navigate to https://httpbin.dev/.

### Expected Output

- Console logs show lines like:  
 >> GET https://httpbin.dev/  
 << 200  
 >> GET https://unpkg.com/@picocss/pico@1.5.7/css/pico.min.css  
 << 200  
- Every network request and response is captured.

## Part B: Assert Valid Status Codes

### Input

Test site: https://httpbin.dev/

### Task

Assert that no responses have status codes outside the 200–299 range.

### Steps

1. Create a `List<Integer>` to collect response status codes.  
2. Attach a `page.onResponse()` listener to add each response’s status code to the list.  
3. Navigate to https://httpbin.dev/.  
4. After navigation, check the list for codes outside 200–299.  
5. Assert that no invalid codes exist.

### Expected Output

- Test passes if all responses return codes within 200–299.  
- If invalid codes exist (e.g., 404), the test fails with:  
 org.opentest4j.AssertionFailedError: Found responses with codes outside 200 range  
 Expected :true  
 Actual :false  
- Example console log: [200, 200, 404]

## Part C: Verify Responses with `.ok()`

### Input

Test site: https://httpbin.dev/

### Task

Use the `.ok()` method to check if responses are successful.

### Steps

1. Create a `List<Response>` to collect responses.  
2. Attach a `page.onResponse()` listener to add responses to the list.  
3. Navigate to https://httpbin.dev/.  
4. Filter the list for responses where `!response.ok()`.  
5. Assert that the filtered list is empty.  
6. If not empty, print a formatted list of failing responses with status code and URL.

### Expected Output

- Test passes if all responses return `.ok() == true`.  
- Test fails with a detailed message if failures are found:  
 Found responses with codes outside 200 range:  
 404: <https://httpbin.dev/js/non-existing-file1.js> **--** **as an example**  
 404: <https://httpbin.dev/js/non-existing-file2.js> **-- as an example**

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