Detailed QA Test Suite Development Plan

Part 1: Web UI Testing with Playwright

1. Environment Setup

Initialize GitHub Repository

Create a new repository, e.g., qa-test-suite.

Clone locally: git clone <repo\_url>

Set up Node.js project

npm init -y

```

Install Playwright and Testing Framework

```bash

npm install @playwright/test

```

Configure Playwright\*\*

- Generate configuration file:

```bash

npx playwright test --init

```

This creates `playwright.config.ts` or `.js` with default settings.

Customize the configuration for parallel execution, retries, reporter formats (e.g., HTML, JSON).

2. Designing Test Cases

- Test Case 1: Page Load Verification

- Verify the page loads successfully and the URL is correct.

- Test Case 2: Element Presence

- Check the presence of the file upload input (`input[type="file"]`).

- Verify the submit button is enabled.

- Test Case 3: File Upload Functionality

- Use `setInputFiles()` to simulate file selection.

- Confirm the file name appears or the upload success message appears.

- Test Case 4: Upload Without Selecting a File

- Attempt to submit without selecting a file and verify error message.

- Test Case 5: Post-Upload Validation

- Confirm the URL or success message indicates upload success.

- Test Case 6: UI Elements & Accessibility

- Verify labels, button texts, and accessibility labels.

3. Implementation steps

- Use `test.beforeEach()` for setup steps common to multiple tests.

- Use `expect()` assertions for validation.

- Capture screenshots on failure for debugging:

```js

test('example', async ({ page }) => {

// test steps

}).afterEach(async ({ page }, testInfo) => {

if (testInfo.status !== testInfo.expectedStatus) {

await page.screenshot({ path: `screenshots/${testInfo.title}.png` });

}

});

```

- Use test data files for uploading different file types or sizes.

4. \*\*Reporting & Continuous Integration

- Configure reporters:

```js

// playwright.config.ts

import { defineConfig } from '@playwright/test';

export default defineConfig({

reporter: [['html'], ['list'], ['json', { outputFile: 'results.json' }]],

retries: 2,

timeout: 30000,

});

```

- Generate HTML reports for visual validation.

- Integrate with CI/CD pipelines if applicable.

Part 2: API Testing with Playwright

1. \*\*Setup & Organization\*\*

- Create `/api` directory inside your project.

- Use Playwright's APIRequestContext for request management.

- Example setup in `playwright.config.ts`:

```js

use: {

baseURL: 'https://jsonplaceholder.typicode.com',

}

```

2. Design API Test Cases

| Endpoint | Method | Test Focus | Example Validation |

|------------|---------|--------------|---------------------|

| GET /posts | GET | Response status, structure | Status 200, array length > 0, contains expected fields |

| GET /posts/{id} | GET | Valid ID returns correct post | Status 200, `id` matches request |

| POST /posts | POST | Create new post | Status 201, response body contains new post data |

| PUT /posts/{id} | PUT | Update existing post | Status 200, updated fields reflect changes |

| DELETE /posts/{id} | DELETE | Delete post | Status 200 or 204, subsequent GET returns 404 |

3. Implementation steps

- Use `test.beforeAll()` to set up common variables.

- Use `request` fixture provided by Playwright for making API calls.

- Example:

```js

import { test, expect } from '@playwright/test';

test.describe('Posts API tests', () => {

const baseURL = 'https://jsonplaceholder.typicode.com';

test('GET /posts returns list', async ({ request }) => {

const response = await request.get('/posts');

expect(response.status()).toBe(200);

const data = await response.json();

expect(Array.isArray(data)).toBeTruthy();

expect(data.length).toBeGreaterThan(0);

});

test('POST /posts creates a new post', async ({ request }) => {

const newPost = {

title: 'foo',

body: 'bar',

userId: 1,

};

const response = await request.post('/posts', { data: newPost });

expect(response.status()).toBe(201);

const responseData = await response.json();

expect(responseData).toMatchObject(newPost);

});

```

- For PUT/PATCH, send updated data and verify the response reflects changes.

- For DELETE, verify response status and optionally check that subsequent GET returns 404 or empty.

4. Error Handling & Edge Cases

- Test invalid IDs (e.g., non-existent IDs).

- Test missing or invalid payloads.

- Check for proper error messages and status codes.

5. Reporting & Logging

- Use Playwright's reporters.

- Log request and response data for debugging.

- Capture response bodies for failed tests.