Homework Assignment for Quality Assurance Engineer Position

Task 1: Manual Testing

TC-ALDI-01 — Add a single product to the shopping list

Objective: Verify a logged-in user can add one product to the Shopping List and see it persist.

Preconditions:

- User account exists and is logged in.
- A local store is selected
- Shopping List is empty.

Test Step	**Action**	**Expected result**
1	From the homepage click on one product	The details of the selected product will appear on a new page.
2	Check the button is available on the page with title "Add to Cart"	The button is available on the page with title "Add to Cart"
3	Click on the "Add to Cart" button.	a delete button, quantity , + button will appear in place of the Add to cart button and the number “1” Shopping List counter will also appear by in the header row next to the cart icon
4	Open the Shopping List panel/page by click on the Cart icon on the header row.	Shopping List displays the selected product with, quantity 1, correct product name, size, and price and subtotal
5	Refresh the page (F5).	The list persists after page refresh and across sessions.
6	Log out and log back in; re-open Shopping List.	The list persists after re-login.

TC-ALDI-02 — Attempt to add a product without being logged in

Objective: Validate how the site behaves when an unauthenticated user tries to add an item to the Shopping List.

Preconditions:

- User logged out

Test Step	Action	Expected result
1	From the homepage click on one product	The details of the selected product will appear on a new page.
2	Check the button is available on the page with title "Add to Cart"	The button is available on the page with title "Add to Cart"
3	Click on the "Add to Cart" button.	user is redirected to ALDI Account page because login required

TC-ALDI-03 — Add multiple products and verify the shopping list

Objective: Verify a logged-in user can add multiple products to the Shopping List, quantities are handled correctly, no duplicated appearence, shopping list stay persist.

Preconditions:

- User account exists and is logged in.
- A local store is selected
- Shopping List is empty.

Test Step	**Action**	**Expected result**
1	From the homepage, search for and open the Product Detail Page of **Product A**	The details of **Product A** appear on a new page.
2	Check the button is available on the page with title **"Add to Cart".**	The button is available on the page with title **"Add to Cart"**
3	Click on the **"Add to Cart"** button for **Product A**.	a delete button, ProductA quantity , + button will appear in place of the "Add to cart" button and the number "1" Shopping List counter will also appear by in the header row next to the cart icon
4	Navigate back to homepage (or search again), search for and open the Product Detail Page of **Product B**	The details of **Product B** appear on a new page.
5	Click on the **"Add to Cart"** button for **Product B**.	a delete button,ProductB quantity , + button will appear in place of the "Add to cart" button and the number "1" C will also appear by in the header row next to the cart icon
6	Increase **Product B** quantity by click on the "+" button.	the ProductB quantity and the Shopping List counter increment by "1"
7	Open the Shopping List panel/page by click on the Cart icon on the header row.	Shopping List displays the selected products with related quantities , correct product name, size, and price and subtotal
8	Refresh the page (F5).	The list persists after page refresh and across sessions.
9	Log out and log back in; re-open Shopping List.	The list persists after re-login.

Bug Reporting:

Title: Duplicate entries appear when adding same product

Build/Version: 2025.08.10 (commit number if version controlled)

Severity: Major
Priority: High
Component: Shopping List – Add Flow

Reporter: Dohos Máté

Description:
When a product is added to the Shopping List from Search Results and then from its PDP, two separate list items appear instead of one item with an incremented quantity. This causes incorrect totals and list clutter.

Steps to Reproduce:

1. Log in, select store and ensure Shopping List is empty.
2. Search ProductA
3. In the search result card, click Add to Cart.
4. Click into the same product's PDP.
5. Add it again by click + button (incrementing quantity)
6. Open the Shopping List.

Expected Result:
One row for ProductA with quantity = 2; correct total and pricing.

Actual Result:
Two identical ProductA rows appear, each with quantity = 1. The list total counts both separately.

Reproducibility: Always

Attachments:

- Screenshot: duplicates_shopping_list.png
- trace log …

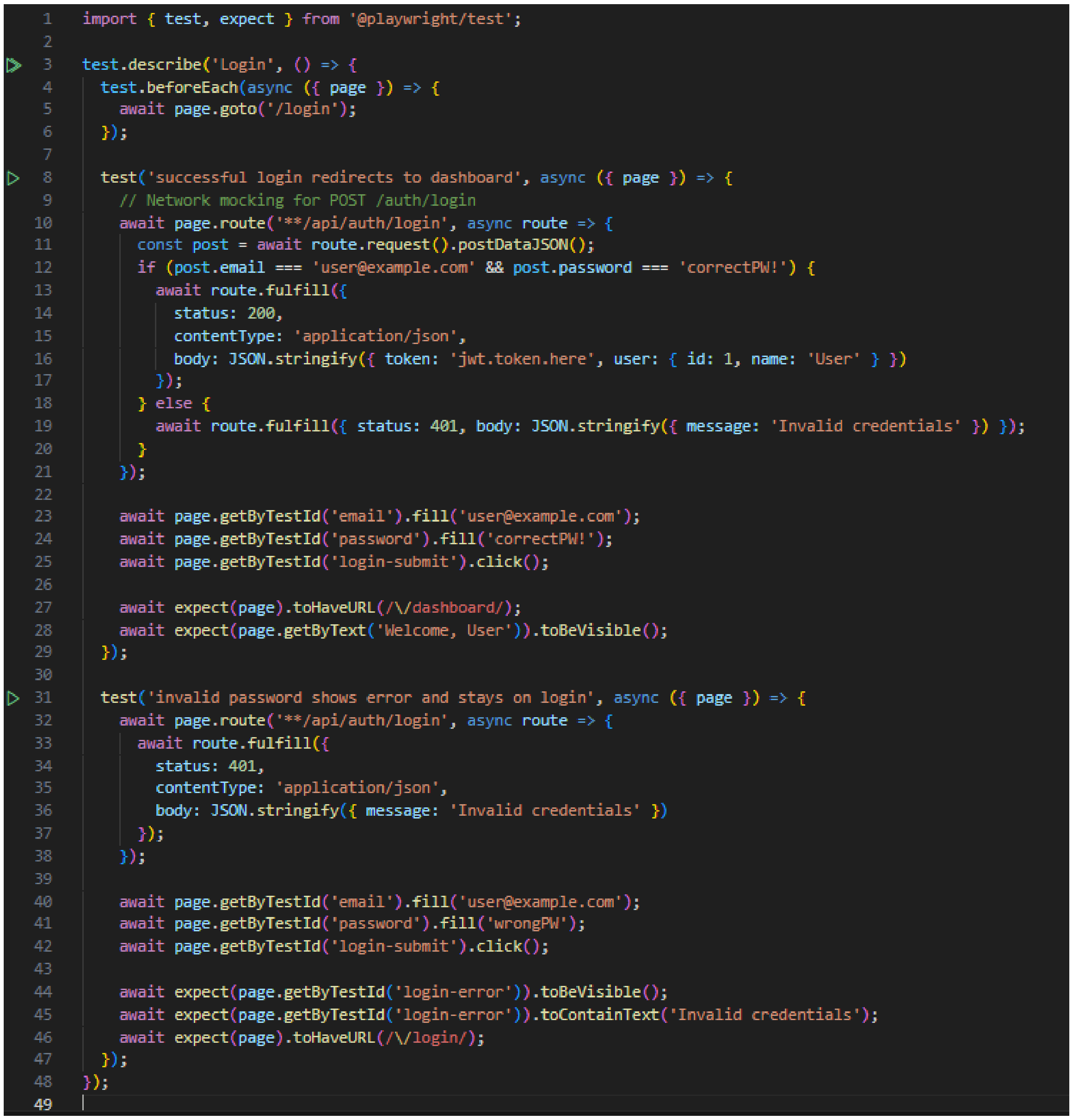
Workaround:
User must manually delete duplicates or adjust quantities.

Additional Notes:
Possible cause: product IDs differ between Search and PDP contexts; normalization missing.

Task 2: Frontend Testing in playwright

The below example shows a usual login process handling in playwright. You can see 1 test for a success login and 1 for a negative in case of invalid pw was given.

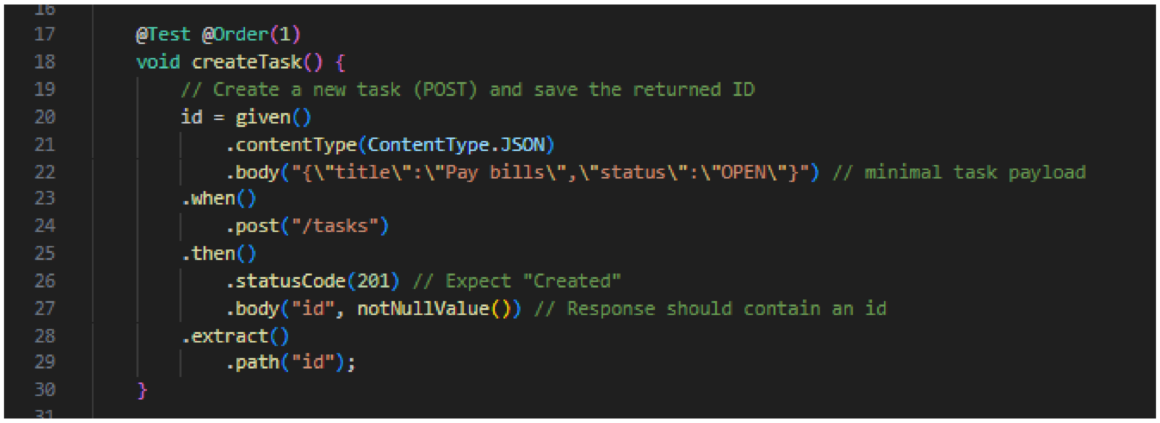
login.spec.ts:



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

test.describe('Login', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('/login');
  });

  test('successful login redirects to dashboard', async ({ page }) => {
    // Network mocking for POST /auth/login
    await page.route('**/api/auth/login', async route => {
      const post = await route.request().postDataJSON();
      if (post.email === 'user@example.com' && post.password === 'correctPW!') {
        await route.fulfill({
          status: 200,
          contentType: 'application/json',
          body: JSON.stringify({ token: 'jwt.token.here', user: { id: 1, name: 'User' } })
        });
      } else {
        await route.fulfill({ status: 401, body: JSON.stringify({ message: 'Invalid credentials' }) });
      }
    });

    await page.getByTestId('email').fill('user@example.com');
    await page.getByTestId('password').fill('correctPW!');
    await page.getByTestId('login-submit').click();

    await expect(page).toHaveURL(/\/dashboard/);
    await expect(page.getByText('Welcome, User')).toBeVisible();
  });

  test('invalid password shows error and stays on login', async ({ page }) => {
    await page.route('**/api/auth/login', async route => {
      await route.fulfill({
        status: 401,
        contentType: 'application/json',
        body: JSON.stringify({ message: 'Invalid credentials' })
      });
    });

    await page.getByTestId('email').fill('user@example.com');
    await page.getByTestId('password').fill('wrongPW');
    await page.getByTestId('login-submit').click();

    await expect(page.getByTestId('login-error')).toBeVisible();
    await expect(page.getByTestId('login-error')).toContainText('Invalid credentials');
    await expect(page).toHaveURL(/\/login/);
  });
});

```

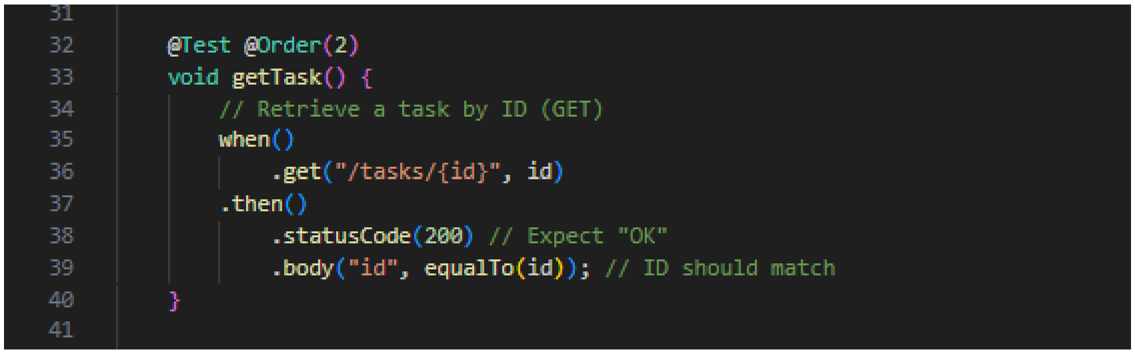
Neverthellest I created a real playwright project for login into aldi.us. This is that you can see next to this docx. The real project contains a README.md where I wrote step by step the project building process. The whole login process doesn't implemented because of the anti-robot-verification.

Task 3: API Testing - Test suite for the endpoints with expected status codes:



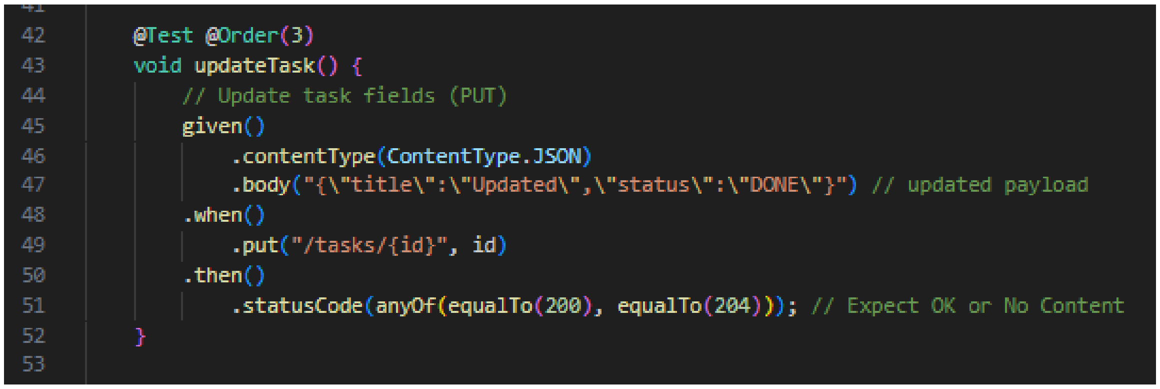
```java
    @Test @Order(1)
    void createTask() {
        // Create a new task (POST) and save the returned ID
        id = given()
            .contentType(ContentType.JSON)
            .body("{\"title\":\"Pay bills\",\"status\":\"OPEN\"}") // minimal task payload
        .when()
            .post("/tasks")
        .then()
            .statusCode(201) // Expect "Created"
            .body("id", notNullValue()) // Response should contain an id
        .extract()
            .path("id");
    }
```

- **POST** → 201 (Created), 400 if invalid.



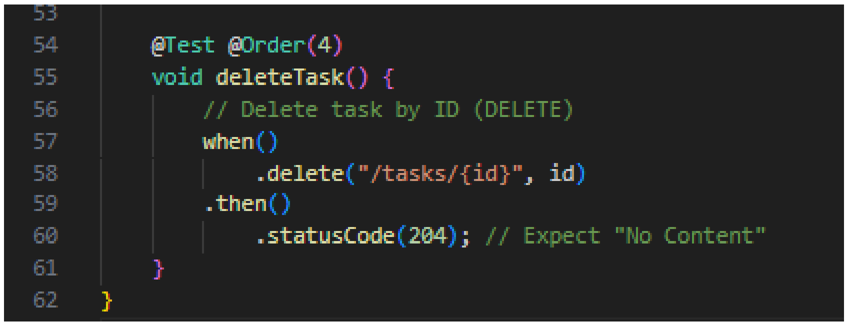
```java
    @Test @Order(2)
    void getTask() {
        // Retrieve a task by ID (GET)
        when()
            .get("/tasks/{id}", id)
        .then()
            .statusCode(200) // Expect "OK"
            .body("id", equalTo(id)); // ID should match
    }
```

- **GET** → 200 (OK), 404 if not found.



```java
    @Test @Order(3)
    void updateTask() {
        // Update task fields (PUT)
        given()
            .contentType(ContentType.JSON)
            .body("{\"title\":\"Updated\",\"status\":\"DONE\"}") // updated payload
        .when()
            .put("/tasks/{id}", id)
        .then()
            .statusCode(anyOf(equalTo(200), equalTo(204))); // Expect OK or No Content
    }
```

- **PUT** → 200 or 204, 400 if invalid, 404 if not found.



```
    @Test @Order(4)
    void deleteTask() {
        // Delete task by ID (DELETE)
        when()
            .delete("/tasks/{id}", id)
        .then()
            .statusCode(204); // Expect "No Content"
    }
}
```

- **DELETE** → 204, 404 if not found.

<u>Bonus Questions</u>

<u>CI Integration:</u>

The following description about how I'd wire Playwright tests into a CI pipeline so they run automatically on every commit and pull request with Jenkins

Environment

CI: Jenkins agents with playwright image

Test runner: @playwright/test- playwright built in runner

Package manager: npm

Artifacts: HTML report, traces, videos, screenshots on failure

Reporting: JUnit + HTML (via Jenkins Test Results and HTML Publisher)

Workflow

1. Trigger on push and PR via Pipeline.
2. Checkout → npm ci → npx playwright install --with-deps
3. Run npx playwright test with retries, headless mode, parallel workers.
4. Archive HTML report and test-results in Jenkins.
5. Publish JUnit results for trend and mark job as required before merge.