Task 1:

1. Identify Flakiness issues:

- Immediately checks page or elements after clicking login.
- No wait for dynamic elements like .welcome-message.
- 2FA (two-factor authentication) not handled.
- No error handling.

2. Explain root causes:

Local Environment	CI/CD Pipeline
Immediate feedback after clicking or load elements	Pages can take longer to redirect
May manually handle OTP locally popup	CI can’t handle unexpected OTP or Screen
Headed browser shows everything clearly rendering	Headless browser might delay

3. Fixes:

```
import pytest

from playwright.sync_api import sync_playwright, expect, TimeoutError

import re


# Reusable login helper

def login(page, email, password):

try:

page.goto("https://app.workflowpro.com/login", timeout=15000)

page.fill("#email", email)

page.fill("#password", password)

page.click("#login-btn")


# Optional: check for 2FA screen

if page.locator("#otp-input").is_visible(timeout=3000):

print("2FA screen detected. Exiting login.")

return False # Signal login was not successful due to 2FA


# Wait for either redirect or dashboard element

expect(page).to_have_url(re.compile(r".*/dashboard"), timeout=10000)
expect(page.locator(".welcome-message")).to_be_visible(timeout=10000)


return True # Login succeeded


except TimeoutError as te:
print(f"[Login TimeoutError] Page took too long to load or element missing: {te}")
return False


except Exception as e:
print(f"[Login Exception] Something went wrong during login: {e}")
return False


@pytest.mark.parametrize("email,password", [
("admin@company1.com", "password123"),
])
def test_user_login(email, password):
with sync_playwright() as p:
browser = None
try:
browser = p.chromium.launch(headless=True, args=["--window-size=1280,720"])
context = browser.new_context(viewport={"width": 1280, "height": 720})
page = context.new_page()


success = login(page, email, password)
```

```
assert success, "Login failed — possibly due to 2FA or timeout"


# Confirm welcome message and dashboard

assert "dashboard" in page.url

assert page.locator(".welcome-message").is_visible()


except AssertionError as ae:

print(f"[AssertionError] {ae}")

raise


except Exception as e:

print(f"[Test Error] An unexpected error occurred: {e}")

raise


finally:

if browser:

browser.close()


@pytest.mark.parametrize("email,password,tenant_keyword", [

("user@company2.com", "password123", "Company2"),

])

def test_multi_tenant_access(email, password, tenant_keyword):

with sync_playwright() as p:

browser = None
```

```
try:

browser = p.chromium.launch(headless=True)

context = browser.new_context(viewport={"width": 1280, "height": 720})

page = context.new_page()


success = login(page, email, password)

assert success, "Login failed — possibly due to 2FA or timeout"


# Wait for project cards to appear

expect(page.locator(".project-card").first).to_be_visible(timeout=10000)


# Check all cards for tenant-specific content

cards = page.locator(".project-card")

for i in range(cards.count()):

content = cards.nth(i).text_content()

assert tenant_keyword in content, f"Tenant data mismatch in card {i}"


except AssertionError as ae:

print(f"[AssertionError] {ae}")

raise


except Exception as e:

print(f"[Test Error] An unexpected error occurred: {e}")

raise
```

```
finally:

if browser:

browser.close()
```

Changes made to the code:

- Reusable Login Function
- Waits Using expect()
- 2FA Detection Logic
- Error Handling with try, except, finally
- Tenant Validation
- Improved URL Assertion
- Test Parameterization with @pytest.mark.parametrize

Task 2:

1. Framework structure:

```
workflowpro-automation/
|
├── tests/                              Contains all test suites
|    ├── ui/                            UI-specific tests
|    ├── api/                           API-specific tests
|   └── e2e/                            End-to-end tests (API + UI)
|
├── page_objects/                       Page Object Model classes
|    ├── base_page.py                   Base class for all pages (common methods)
|    ├── login_page.py
|   └── dashboard_page.py
|
├── utils/                              Reusable utilities and helpers
|    ├── api_client.py                  Wrapper for making API requests
|    ├── config_manager.py              Handles loading env/browser configs
|   └── browserstack_manager.py         Helper for BrowserStack integration
|
├── config/                             Configuration files
|    ├── base_config.json               Default settings
|    ├── company1_staging.json          Tenant 1 on staging
|    ├── company2_staging.json          Tenant 2 on staging
|   └── production.json                 Production environment config
```

```
|
 ├── test_data/                         Test data files
|   └── user_roles.json                 Data for different user permissions
|
 ├── fixtures/                          Custom pytest fixtures
|   └── browser_setup.py                Fixture for page object, API client, etc.
|
 ├── reports/                           Test execution reports
|
 ├── .env.example                       Example for environment variables
 ├── pytest.ini                         Pytest configuration file
└── requirements.txt                    Project dependencies
```

Tests: This folder contains all the actual test cases. These are the Python files that verify the application's functionality.

Page Objects: In this folder, each page or major component of the app (like Login, Dashboard, Project Form) gets its own Python class. This class holds the locators (like CSS selectors) for all elements on that page and methods to interact with them.

Utils: This is a toolbox for your framework. It holds reusable helper scripts and classes that aren't tied to one specific page.

Config: This folder is for **environment settings**. It defines *where* and *how* the tests should run.

Test Data: This folder is for data that the tests **use as input**. This separates the data from the config.

Fixtures: A dedicated place for shared pytest fixtures.

Reports: This is where you save the results after tests run.

.env.example: This is a template file that shows what environment variables are needed to run the project.

pytest.ini: A central configuration file for pytest. used to define custom markers, set default command-line options, and control test discovery paths.

2. Configuration management:

config/ folder: This folder holds settings that define the environment where your tests will run. You'll have a base file for common settings and specific files for each environment (like staging or production). This avoids duplicating settings.

- base_config.json: Contains default values that apply to all environments.
- staging.json: Contains values that override the base settings specifically for the staging environment.
- production.json: Contains overrides for the production environment.

test_data/ folder: This folder holds data that your tests will *use* as input, but it's not related to the environment itself.

- user_roles.json: Contains information about different types of users (Admin, Manager, Employee) and their permissions.
- project_payloads.json: Could contain sample JSON bodies for creating different kinds of projects via the API.

.env file: This is a single file at the root of your project that holds all sensitive information. Crucially, this file should never be committed to source control.

- ADMIN_PASSWORD="password123"
- BROWSERSTACK_ACCESS_KEY="your-secret-key"
- API_TOKEN="your-api-token"

3. Identify missing requirements:

Test data management:

- Is the data static (in JSON files) or dynamic
- How will test data be managed for different environments Reporting:
- What kind of reports are required? (HTML, Allure, ect) Should reports include screenshots of failures?

Parallel Execution:

- Do we need to run tests in parallel?
- Can the environment handle concurrent requests? What are the tools used for parallel execution?

CI/CD Integration:

- Which CI tool are we using?
- Should tests run on every pull request or only on main branch?
- Where should reports and logs be saved in CI?

Task 3:

This test validates the end-to-end flow of creating and verifying a project across API, web UI, mobile UI, and different tenants for isolation.

1. API: Create Project

    Tool used: requests + playwright

2. Web UI: to check project visibility

    Tool used: playwright

3. Mobile UI: mobile browser check

    Tool used: BrowserStack + playwright

4. Isolation: Ensure Security

    Tool used: API

```
import pytest

import uuid

from playwright.sync_api import Page, expect

from utils.api_client import APIClient # Assumed custom API client


# Define test data for two different tenants

TENANT_1 = {

  "id": "company1",

  "user": "manager@company1.com",

  "pass": "password123"

}

TENANT_2 = {
  "id": "company2",
  "user": "user@company2.com",
  "pass": "password456"
}


@pytest.fixture(scope="function")
def api_client_t1():
  """Fixture to provide an authenticated API client for Tenant 1."""
  client = APIClient(tenant_id=TENANT_1["id"])
  client.authenticate(TENANT_1["user"], TENANT_1["pass"])
  return client


@pytest.fixture(scope="function")
def project_data_t1(api_client_t1):
  """
  Sets up test data by creating a project via API and handles cleanup.
  This is a core principle: use APIs for fast setup/teardown.
  """
  project_name = f"Project-{uuid.uuid4()}"
  project_payload = {
      "name": project_name,
      "description": "An E2E test project."
  }
  # Setup
  response = api_client_t1.create_project(project_payload)
  assert response.status_code == 201
  project = response.json()

  # Pass project data to the test
  yield project

  # Teardown
  print(f"Cleaning up project ID: {project['id']}")
  cleanup_response = api_client_t1.delete_project(project['id'])
  assert cleanup_response.status_code in [200, 204]

def test_project_creation_and_isolation_flow(page: Page, project_data_t1):
  """
  End-to-end test for project creation and tenant isolation.
  1. API: Create project for Company1 (done in fixture).
  2. Web UI: Verify project is visible for Company1 user.
  3. Mobile: Verify project is visible on a mobile browser view.
  4. Security: Verify project is NOT visible to a Company2 user.
  """

  # 2. Web UI: Verify project display for Company1
  # Login as a user from the correct tenant (Company1)
```

```
page.goto("/login")

page.get_by_placeholder("Email").fill(TENANT_1["user"])

page.get_by_placeholder("Password").fill(TENANT_1["pass"])

page.get_by_role("button", name="Login").click()


# Navigate to the projects page and verify the new project is visible

expect(page).to_have_url(re.compile(r".*/dashboard"))


# This is a robust way to find the element using the unique name from the API response

new_project_card = page.locator(f".project-card:has-text('{project_data_t1['name']}')")

expect(new_project_card).to_be_visible(timeout=10000)

print(f"SUCCESS: Project '{project_data_t1['name']}' is visible in UI for Tenant 1.")


# 3. Mobile: Check mobile accessibility

# This part demonstrates the thinking for BrowserStack integration.

# In a real framework, this would be abstracted.

# Here, we can simulate it by changing the viewport. A full BrowserStack

# implementation would involve a remote browser context.


print("Simulating mobile view check...")

page.set_viewport_size({"width": 390, "height": 844}) # iPhone 13 viewport

page.reload() # Reload to get the mobile layout


# Re-verify the project is visible and rendered correctly in the mobile layout
```

```
mobile_project_card = page.locator(f".project-card:has-text('{project_data_t1['name']}')")

expect(mobile_project_card).to_be_visible()

print(f"SUCCESS: Project '{project_data_t1['name']}' is visible in mobile viewport.")


# Reset viewport for subsequent steps

page.set_viewport_size({"width": 1920, "height": 1080})


# Logout to switch users

page.get_by_role("button", name="Logout").click()

expect(page).to_have_url(re.compile(r".*/login"))


# 4. Security: Verify tenant isolation

# Login as a user from a different tenant (Company2)

page.get_by_placeholder("Email").fill(TENANT_2["user"])

page.get_by_placeholder("Password").fill(TENANT_2["pass"])

page.get_by_role("button", name="Login").click()


expect(page).to_have_url(re.compile(r".*/dashboard"))


# Assert that the project created for Company1 is NOT visible to Company2 user.

# This is a critical security check.

isolated_project_card=page.locator(f".project-card:has-text('{project_data_t1['name']}')")

expect(isolated_project_card).to_be_hidden() # or to_have_count(0)

print(f"SUCCESS: Project '{project_data_t1['name']}' is NOT visible for Tenant 2.")
```

Test Structure:

Step 1: Create a project using the API

- This is handled by a pytest fixture called `project_data_t1`.
- The fixture uses a helper class (`APIClient`) to send a POST request to the /api/v1/projects endpoint.
- It automatically includes the `Authorization: Bearer {token}` and `X-Tenant-ID: company1` headers.
- The JSON body includes a unique, dynamically generated project name.
- It extracts the full project object (including 'id' and 'name') from the
- API response and passes it to the test.
- Crucially, the fixture also handles cleanup by sending a DELETE request after the test is finished, ensuring a clean state.

Step 2: Verify project appears in the Web UI

- Use Playwright's `page` object to log into the web app as a user from the same tenant ("manager@company1.com").
- Navigate to the dashboard page.
- Assert that the project is visible by using the unique `name` from the API response in

Step 1.

Example: `page.locator(f".project-card:has-text('{project_data_t1['name']}')")`

- Use `expect(element).to_be_visible()` to wait for the element to appear, which prevents flaky tests from failing on slow-loading pages.

Step 3: Verify project is accessible on Mobile UI (BrowserStack)

- To conceptually test this, we can simulate a mobile device by resizing the browser viewport with Playwright.

Example: `page.set_viewport_size({"width": 390, "height": 844})`

- After resizing, the test reloads the page and runs the same verification as in Step 2 to ensure the project is still visible in the mobile layout.

- A full implementation would connect to a remote BrowserStack device, but this demonstrates the validation of responsive design.

Step 4: Validate tenant isolation

- Log out of the application.

- Log back in as a user from a *different* company/tenant

    (e.g., "user@company2.com").

- Navigate to the dashboard.

- Assert that the project created for Company1 is **NOT** visible.

Example: `expect(isolated_project_card).to_be_hidden()`

- This confirms that one tenant's data is not accessible to another, which is a critical security boundary.

pass