Playwright with python Framework steps

Step 1: Install Dependencies

```
pip install playwright pytest pytest-html pytest-xdist
```

```
playwright install
```

- **playwright** → for automation
- **pytest** → to run tests easily
- **pytest-html** → to generate HTML reports
- **pytest-xdist** → to run tests in parallel (optional)
- `playwright install` → downloads browsers (Chrome, Firefox, etc.)

Step 2: Config File (`utils/config.py`)

```
BASE_URL = "https://mymoose.wd1.myworkdayjobs.com/en-US/careers/login?redirect=%2Fen-US%2Fcareers%2FuserHome"
USERNAME = "student"
PASSWORD = "Password@123"
INVALID_EMAIL = "student@gmail.com"
```

This file keeps **settings and test data** in one place.
If tomorrow the URL or password changes, you don't touch the test code — just update here.

Step 3: Page Object Model (POM)

```
from playwright.sync_api import Page, expect

class LoginPage:
    def __init__(self, page: Page):
        self.page = page
        self.email_input = page.get_by_role("textbox", name="Email Address")
        self.password_input = page.get_by_role("textbox", name="Password")
        self.signin_button = page.get_by_role("button", name="Sign In")
        self.error_message = page.get_by_text("You may have entered the")

    def open(self, url: str):
        self.page.goto(url)

    def login(self, username: str, password: str):
        self.email_input.fill(username)
        self.password_input.fill(password)
        self.signin_button.click()

    def assert_error_message_visible(self):
        expect(self.error_message).to_be_visible(timeout=1000)
```

Instead of writing locators & actions directly in test, we create a **class for each page** (Login page, Home page, etc.).
Benefits:

- Code is **reusable**
- Tests look **cleaner** (like reading English: `login.login(username, password)`)

Step 4: Fixtures (`conftest.py`)

```
import pytest
from playwright.sync_api import sync_playwright

@pytest.fixture(scope="session")
def browser():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=False)
        yield browser
        browser.close()

@pytest.fixture(scope="function")
def page(browser):
    context = browser.new_context()
    page = context.new_page()
    yield page
    context.close()
```

Fixtures are like **setup & cleanup** steps.

- `browser` fixture → opens browser **once per session**
- `page` fixture → opens new page (tab) **for each test**, and closes it afterwards

So you don't repeat `launch browser` in every test.

Step 5: Test File (`tests/test_login.py`)

```
from pages.login_page import LoginPage
from utils import config

def test_invalid_login(page):
    login = LoginPage(page)
    login.open(config.BASE_URL)
    login.login(config.USERNAME, config.PASSWORD)
    # Enter invalid email second time
    login.email_input.fill(config.INVALID_EMAIL)
    login.signin_button.click()
    login.assert_error_message_visible()
```

This is the **actual test case**.

- We create `LoginPage` object
- Call `login.open()` → opens the login URL
- Call `login.login()` → enters user & password
- Call `login.assert_error_message_visible()` → checks if error shows up

Notice how **short & clean** this looks — because logic is inside Page Object.

Step 6: Pytest Config (`pytest.ini`)

```
```

```
[pytest]
addopts = -v --html=report.html --self-contained-html
```

Tells pytest:

- `-v` → show detailed output
- `--html=report.html` → generate an HTML report
- `--self-contained-html` → makes report portable (with styles/images inside)

Step 7: Run Tests

```
Pytest
```

This command:

- Opens browser
- Runs test(s)
- Closes browser
- Creates a **report.html** file

Now you have:

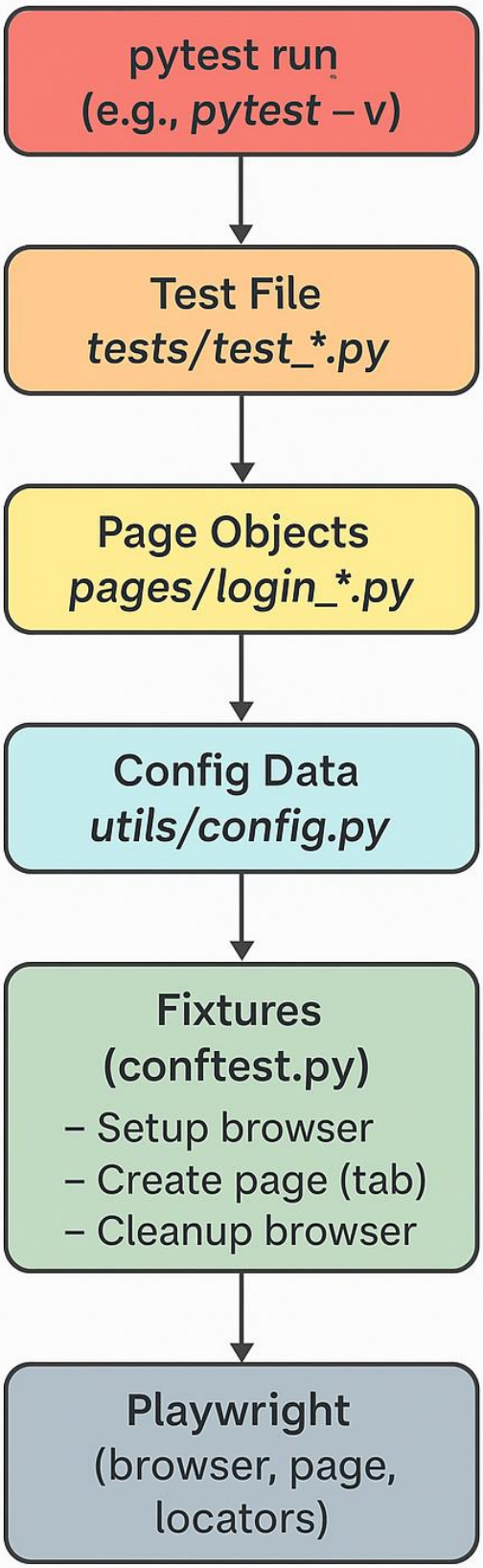
- A **framework structure** (like a house foundation)
- Page Object Model for cleaner tests
- Reusable configs
- Auto setup/teardown with fixtures
- HTML report

How It Works Step by Step

1. **pytest starts** → finds all `test_*.py` files inside `tests/`.
2. **Test File (`test_login.py`)** → says: "I want to login."
3. It calls **Page Object (`login_page.py`)** → which has locators & actions like `login()`.
4. The Page Object uses **Config (`config.py`)** → to fetch URL, username, password.
5. Test automatically gets **page/browser from fixtures (`conftest.py`)**.
6. Playwright runs the **actual browser automation**.
7. After test → browser & context are closed by fixtures.
8. pytest creates an **HTML report** with results.

Tip: Think of it like a company:

- **Test file = Manager** (just says what to do)
- **Page Object = Employee** (knows *how* to do it)
- **Config = Office handbook** (rules, credentials, data)
- **Fixtures = Office admin** (sets up computers, cleans up after work)
- **Playwright = Machine** (executes the actual work)


pytest run
(e.g., pytest – v)
Test File
tests/test_*.py
Page Objects
pages/login_*.py
Config Data
utils/config.py
Fixtures
(conftest.py)
– Setup browser
– Create page (tab)
– Cleanup browser
Playwright
(browser, page,
locators)

Feature / Task	Selenium (Java Example)	Playwright (Python Example)	Key Difference
Setup	`WebDriver driver = new ChromeDriver(); driver.get("https://example.com");`	`from playwright.sync_api import sync_playwright \nwith sync_playwright() as p:\n browser = p.chromium.launch()\n page = browser.new_page()\n page.goto("https://example.com")`	Selenium needs external driver setup; Playwright bundles browsers.
Finding Elements	`driver.findElement(By.id("username")).sendKeys("test");`	`page.locator("#username").fill("test")`	Playwright uses `locator()` API, more powerful than `findElement`.
Click Action	`driver.findElement(By.name("login")).click();`	`page.get_by_role("button", name="Login").click()`	Playwright encourages role/text-based locators → less brittle.
Assertions	`Assert.assertEquals(driver.getTitle(), "Dashboard");`	`from playwright.sync_api import expect expect(page).to_have_title("Dashboard")`	Playwright has assertions built-in (no TestNG/JUnit).
Waits	`WebDriverWait wait = new WebDriverWait(driver, 10);\nwait.until(ExpectedConditions.visibilityOfElementLocated(By.id("welcome")));`	`page.locator("#welcome").click() # auto-waits`	Playwright auto-waits for elements → fewer flaky tests.
Handling Alerts	`Alert alert = driver.switchTo().alert(); alert.accept();`	`page.on("dialog", lambda dialog: dialog.accept())`	Playwright uses event listeners for dialogs.
Multiple Tabs / Windows	`String parent = driver.getWindowHandle(); for(String h: driver.getWindowHandles()) { driver.switchTo().window(h); }`	`new_page = page.context.new_page()\nnew_page.goto("https://example.com")`	Playwright directly creates new pages (tabs).
Frames / Iframes	`driver.switchTo().frame("frameName");`	`frame = page.frame(name="frameName")\nframe.locator("#btn").click()`	Direct frame reference, no context switching.
Screenshots	`File scr = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);`	`page.screenshot(path="screenshot.png")`	One-liner in Playwright.
Headless Mode	`ChromeOptions options = new ChromeOptions();\noptions.addArguments("--headless");`	`browser = p.chromium.launch(headless=True)`	Simpler in Playwright.

Feature / Task	Selenium (Java Example)	Playwright (Python Example)	Key Difference
Parallel Execution	Needs TestNG/JUnit + Selenium Grid/Docker.	`bash pytest -n 4 --browser chromium` (with pytest-xdist)	Playwright has native parallelism in runner.
Reporting	Needs ExtentReports / Allure.	`bash pytest --html=report.html` (pytest-html) or Playwright built-in HTML report.	Playwright has reports built-in.
Network Mocking / API Testing	Needs BrowserMob Proxy or external libs.	`page.route("**/api/*", lambda route: route.fulfill(body='{"status":"mocked"}'))`	Playwright can mock network requests directly.
Debugging	Logs & IDE debugger.	`bash pytest --headed --slowmo 500` or Trace Viewer: `playwright show-trace trace.zip`	Playwright offers full visual test replay.