pytest-playwright - week-1,5-day learning plan

Nice — love the focus. Here's a **teacher-style, hands-on 5-day plan** for *Week 1 — Foundations: Selectors, Assertions & Core Interactions* using **Playwright + Pytest**. Each day has a step-by-step practice plan, short code examples you can copy, and **15–20 interview-style questions with concise answers** (theory + conceptual). Work 60–120 minutes/day if you can — do the exercises and then run the interview questions out loud or on a whiteboard.

Quick setup (before Day 1)

1. Install Playwright and Pytest:
   `pip install pytest playwright`
   `playwright install`
   (or use **`pip install pytest-playwright`** and follow that plugin's docs).
2. Create a project folder, a **`tests/`** directory and a **`conftest.py`** for fixtures (example below).
3. Pick practice sites: Playwright Demo (**`https://playwright.dev/playwright-test/`**) and Rahul Shetty Automation Practice (forms/tables).

Common fixture (single-file example)

Put this in **`tests/conftest.py`** (sync example). You can also use the pytest-playwright plugin which already provides **`page`** and **`browser`** fixtures.

```python
# conftest.py
import pytest
from playwright.sync_api import sync_playwright

@pytest.fixture(scope="session")
def browser_context():
    pw = sync_playwright().start()
    browser = pw.chromium.launch(headless=False)  # headless=True in CI
    context = browser.new_context()
    yield context
    context.close()
    browser.close()
    pw.stop()

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

Use **`page`** in tests: **`def test_example(page): page.goto("https://demo.playwright.dev") ...`**

DAY 1 — Launching browsers, Pytest fixtures, page vs context

Goal

Understand launching browsers from pytest, fixtures lifecycle, and difference between **`page`** and **`context`**.

Step-by-step practice

1. Read/skim Playwright fixture concept (locally): what **`browser`**, **`context`**, **`page`** represent.
2. Create the **`conftest.py`** above and run **`pytest -q`** to see fixture usage.
3. Make a simple test:

   ```python
   def test_open_home(page):
       page.goto("https://playwright.dev")
       assert "Playwright" in page.title()
   ```

4. Experiment: open multiple pages in same context (simulate multiple tabs). Then create a new context for isolated cookies/storage.
5. Try launching headless vs headed and observe performance.

Exercises

- Write tests that open two pages in same context and assert cookies are shared.
- Write tests using two different contexts to show isolation.

18 Interview Qs (Day 1)

1. **Q: What is Playwright?**
   A: A browser automation library for Chromium/Firefox/WebKit supporting modern web features and cross-browser automation.
2. **Q: What's the difference between `browser`, `context`, and `page`?**
   A: **`browser`** = the browser process; **`context`** = an isolated browser session (cookies, localStorage); **`page`** = a single tab/frame within a context.
3. **Q: Why use multiple contexts?**
   A: To isolate sessions (different users, different storage/cache); useful in parallel tests.
4. **Q: When to reuse a context vs create a new one?**
   A: Reuse for speed if tests can share state. Create new to avoid state bleed between tests.
5. **Q: What is a Pytest fixture and why use it with Playwright?**
   A: A fixture provides setup/teardown for tests (e.g., creating browser/page). It makes tests clean and reusable.
6. **Q: What fixture scope options exist and when to use them?**
   A: **`function`**, **`class`**, **`module`**, **`session`** — use **`session`** for expensive setup like browser launch in CI, **`function`** for isolated pages.
7. **Q: How do you run Playwright tests headless?**
   A: Launch the browser with **`headless=True`** or set env var. In pytest-playwright, set config or use CLI flags.

8. **Q: How to capture console logs from a page?**
   A: Listen to `page.on("console", handler)` and collect messages.
9. **Q: What are advantages of `new_context()`?**
   A: Fast session-level isolation without launching a new browser.
10. **Q: Can you run tests in parallel?**
    A: Yes with pytest-xdist and separate contexts (or playwright test runner). Ensure tests are isolated.
11. **Q: What's Playwright's default timeout?**
    A: Typically 30 seconds; adjustable via options like `page.set_default_timeout()`.
12. **Q: How to debug a failing test locally?**
    A: Run headed (`headless=False`), add `page.pause()`, use `playwright codegen`, or add `page.screenshot()`.
13. **Q: What happens if you don't close a context?**
    A: Resource leak: browser may hold memory and cause flakiness or exhaustion.
14. **Q: Why prefer contexts over separate browser instances?**
    A: Contexts are lighter weight and faster to create than full browser processes.
15. **Q: How do you persist storage across contexts?**
    A: Use `storage_state` option to save and reuse cookies/localStorage.
16. **Q: What is `storage_state`?**
    A: JSON file with cookies and localStorage that can be used to create authenticated contexts.
17. **Q: How to test file downloads in Playwright?**
    A: Use `page.expect_download()` context manager and then `download.path()`.
18. **Q: What is the Playwright CLI `playwright install` for?**
    A: Installs browser binaries required for automation.

DAY 2 — Locators: `get_by_role`, `locator`, `nth()`, CSS & XPath

Goal

Master locating elements reliably: prefer semantic locators (`get_by_role`) when possible; fallback to CSS/XPath when necessary.

Step-by-step practice

1. Learn `page.get_by_role()` for accessible names (buttons, links, headings). Try `page.get_by_role("button", name="Submit")`.
2. Practice `page.locator("css")` and chaining: `page.locator("ul.todo-list li").nth(0)`.
3. Use `nth()` to pick particular items.
4. Practice `locator.filter()` and `locator.locator()` chaining.
5. Try XPath: `page.locator("//div[@class='x']")` — keep XPath as fallback.
6. Use `get_by_text()` and `get_by_placeholder()` for inputs.
7. Correlate locators with devtools selectors and accessible tree.

Short examples

```python
python

# find by role
page.get_by_role("button", name="Add").click()

# CSS locator and nth
first_todo = page.locator("ul.todo-list li").nth(0)
expect(first_todo).to_have_text("Buy milk")

# XPath fallback
page.locator("//input[@id='email']").fill("a@b.com")
```

20 Interview Qs (Day 2)

1. **Q: What is `locator()`?**
   A: A Playwright handle representing one or many elements; methods on it act with auto-waiting.
2. **Q: Why prefer `get_by_role()`?**
   A: It's robust, accessibility-aware, and less brittle to UI changes.
3. **Q: What does `nth()` do?**
   A: Picks the element at the given zero-based index from a matched set.
4. **Q: Difference between `locator(...).first()` and `.nth(0)`?**
   A: Both pick first; `first()` is a convenience; `.nth()` accepts any index.
5. **Q: How does Playwright handle waiting with locators?**
   A: Locator actions auto-wait for element to be actionable (visible, enabled).
6. **Q: When to use XPath vs CSS?**
   A: Use CSS for performance and simplicity; XPath for complex ancestor/sibling relationships.
7. **Q: How to locate an element by placeholder text?**
   A: `page.get_by_placeholder("Search...")` or CSS `input[placeholder="Search..."]`.
8. **Q: What is `get_by_text()`?**
   A: Finds element by visible text (case and substring behaviours configurable).
9. **Q: Are locators evaluated immediately?**
   A: No — locators are lazy; they resolve at action time with auto-waits.
10. **Q: How to scope a locator inside another?**
    A: `parent.locator("child-selector")` or `page.locator("parent").locator("child")`.
11. **Q: What is `has_text` or `has` filter?**
    A: Options to filter locators by nested text or child element.
12. **Q: How to debug a locator that returns multiple elements?**
    A: Use `locator.count()` or `locator.all_text_contents()` to inspect.
13. **Q: When are CSS selectors brittle?**
    A: When they rely on classes or structure likely to change; prefer semantic selectors.
14. **Q: How to wait for an element to appear without action?**
    A: `page.wait_for_selector("selector")` or `expect(locator).to_be_visible()`.

15. **Q: How to select by ARIA attributes?**
    A: Use `get_by_role()` and `get_by_label()` which leverage ARIA labels.
16. **Q: Can you combine role and text?**
    A: Yes: `page.get_by_role("button", name="Save settings")`.
17. **Q: What if multiple elements have same role and name?**
    A: Use additional filters like `locator.nth()` or scope within parent.
18. **Q: How to handle dynamic attributes (IDs change each run)?**
    A: Use stable attributes (data-test-id), roles, labels or relative locators.
19. **Q: Difference between `page.query_selector` and `locator`?**
    A: `query_selector` returns an element handle immediately and you must manage waits; `locator` is higher-level and auto-waits.
20. **Q: What are `data-*` attributes for testing?**
    A: `data-test-id` are stable test attributes to avoid brittle selectors.

DAY 3 — Interactions: click, type, checkboxes, radio, keyboard

Goal

Perform user interactions reliably and learn how Playwright ensures actions are safe.

Step-by-step practice

1. Practice `locator.click()`, `locator.fill()`, `locator.type()` differences:
    - `fill()` replaces; `type()` emulates typing.
2. Work with checkboxes: `locator.check()`, `locator.uncheck()`, `locator.is_checked()`.
3. Radio buttons: `locator.check()` or `page.locator("input[type=radio]").check()` then assert.
4. Keyboard actions: `page.keyboard.press("Enter")`, `page.keyboard.type("abc")`.
5. Combined flows: focus input, type, press Enter, wait for result.
6. Practice clicking dynamic elements (ensure they're visible/enabled) and retry patterns with `expect`.

Examples

```python
page.locator("input#name").fill("Ram")
page.locator("button", has_text="Submit").click()
page.keyboard.press("Tab")
page.locator("input[type=checkbox]#agree").check()
assert page.locator("input[type=checkbox]#agree").is_checked()
```

20 Interview Qs (Day 3)

1. **Q: Difference between `fill()` and `type()`?**
   A: `fill()` sets the whole value (fast), `type()` emulates keystrokes.
2. **Q: Why use `locator.click()` over JS `click()`?**
   A: Locator click auto-waits and ensures element is actionable (visible, enabled).
3. **Q: How to upload a file?**
   A: `page.set_input_files("input[type=file]", "/path/file")`.
4. **Q: How to handle double click or right click?**
   A: Use `locator.dblclick()` or `locator.click(button='right')`.
5. **Q: How do you check a checkbox only if not checked?**
   A: Use `locator.check()` which is idempotent (it checks only if needed).
6. **Q: How to press combination keys (Ctrl+S)?**
   A: `page.keyboard.press("Control+S")`.
7. **Q: Is `locator.click()` synchronous or async?**
   A: In sync API, it blocks until action completes; underlying it awaits promises.
8. **Q: How to move mouse or hover?**
   A: `locator.hover()` or `page.mouse.move(x,y)`.
9. **Q: How to clear input content?**
   A: `locator.fill("")` clears, or `locator.press("Control+A")` then `Backspace`.
10. **Q: Can Playwright type into contentEditable elements?**
    A: Yes, `locator.fill()/type()` work with contentEditable as well.
11. **Q: What if click is intercepted by another element?**
    A: Playwright retries or throws — you'd wait for overlay to disappear or click an alternative locator.
12. **Q: How to emulate slow typing?**
    A: Use `locator.type("text", delay=100)` to set delay between keystrokes.
13. **Q: How to check keyboard focus?**
    A: Evaluate `page.evaluate("document.activeElement")` or check class/attribute.
14. **Q: How do you assert that a button is enabled before clicking?**
    A: `expect(locator).to_be_enabled()`.
15. **Q: How to handle dropdowns that are not `<select>`?**
    A: Click the control, then click the option by text or index.
16. **Q: Are interactions auto-waited?**
    A: Yes — Playwright waits for element readiness (visible/enabled).
17. **Q: How to perform drag & drop?**
    A: `locator.drag_to(target_locator)`.
18. **Q: How to simulate paste?**
    A: `page.keyboard.down("Control"); page.keyboard.press("V"); page.keyboard.up("Control")` or use clipboard APIs.
19. **Q: Why avoid using `time.sleep()` for waits?**
    A: It's flaky and slows tests; prefer Playwright waits and expectations.

20. **Q: How to assert element receives an event (like click triggered function)?**
    A: Check for side effects (DOM change, network call) or use `page.on("request")` to observe network.

DAY 4 — Dropdowns, Alerts/Dialogs, Waits & Assertions

Goal

Handle selects, browser dialogs, and write robust waits and assertions.

Step-by-step practice

1. Native `<select>`: use `locator.select_option(value="1")`, assert selected with `locator.input_value()` or `locator.evaluate`.
2. Custom dropdowns: click opener, click option by text or role.
3. Dialogs (alert/confirm/prompt): use `page.on("dialog", handler)` or `page.expect_dialog()` to assert message and accept/dismiss.

```
python

with page.expect_dialog() as dlg:
    page.click("button#trigger-alert")
dialog = dlg.value
assert dialog.message == "Are you sure?"
dialog.accept()
```

4. Waits:
    - `page.wait_for_selector()`
    - `expect(locator).to_be_visible()`
    - `expect(locator).to_have_text("...")`
    - `page.wait_for_load_state("networkidle")`
5. Assertion styles:
    - Built-in `assert` vs Playwright `expect()` — `expect` has built-in retries and better messages.

Exercises

- Automate a flow with a native select and a custom dropdown.
- Trigger an `alert`, assert text and accept.
- Create a failing assertion and fix it with appropriate wait.

20 Interview Qs (Day 4)

1. **Q: How to select an option in `<select>`?**
   A: `locator.select_option(value="val")` or by label/index.

2. **Q: How to handle JS `alert()` or `confirm()`?**
   A: Use `page.expect_dialog()` to capture and `dialog.accept()` or `dialog.dismiss()`.
3. **Q: What's `expect(locator).to_have_text()` vs `assert`?**
   A: `expect` retries until timeout; `assert` checks immediately.
4. **Q: When use `wait_for_load_state("networkidle")`?**
   A: When you need to wait for background requests to finish (network quiet).
5. **Q: How to wait for element to be hidden?**
   A: `expect(locator).to_be_hidden()` or `page.wait_for_selector(selector, state="hidden")`.
6. **Q: What are common `wait_for_selector` states?**
   A: `attached`, `detached`, `visible`, `hidden`.
7. **Q: Why avoid implicit waits?**
   A: Playwright uses explicit waits; implicit waits can be less predictable.
8. **Q: How to assert attribute value?**
   A: `expect(locator).to_have_attribute("aria-checked","true")`.
9. **Q: How to wait for network requests?**
   A: Use `page.wait_for_response()` or `page.expect_request()` depending on version.
10. **Q: How to handle delayed UI changes?**
    A: Use `expect` with a timeout or `wait_for_selector` for the change condition.
11. **Q: How to assert that an element is enabled?**
    A: `expect(locator).to_be_enabled()`.
12. **Q: How to test prompts that require input?**
    A: Use `page.expect_dialog()` then `dialog.accept("text")`.
13. **Q: What is `networkidle`?**
    A: A load state where there are no network requests for at least 500ms (Playwright semantics).
14. **Q: How to assert element count?**
    A: `expect(locator).to_have_count(n)`.
15. **Q: How to wait for animation to finish?**
    A: Wait for element detached/visible or use `page.wait_for_timeout()` (rare), better to wait for final state.
16. **Q: What's a flakey test and how to reduce flakiness?**
    A: Tests that sometimes pass/fail due to timing. Use robust selectors, waits, and stable test data.
17. **Q: How to assert text contains substring?**
    A: `expect(locator).to_contain_text("partial")`.
18. **Q: Difference `to_be_visible()` vs `to_be_attached()`?**
    A: `visible` means in DOM and visible to user; `attached` just in DOM regardless of visibility.
19. **Q: How to intercept and mock network responses?**
    A: Use `route/page.route()` to intercept and fulfill with mocked responses.
20. **Q: How to verify navigation happened to a new URL?**
    A: `expect(page).to_have_url("https://...")` or `page.wait_for_url()`.

DAY 5 — Putting it together: Mini projects & traces/screenshots

Goal

Complete four mini projects from your Week 1 list; capture screenshots/traces; write tidy tests.

Mini Projects (step-by-step)

Project A — Add & verify items in Playwright Todo App

1. `page.goto("playwright todo url")`
2. Add three todo items (use `fill` + `press("Enter")` or click Add).
3. Assert items count with `locator.count()`.
4. Mark one complete, assert strike-through or class change.

Project B — Automate form fill + assert success message (Rahul Shetty form)

1. Locate inputs by `get_by_label` or `get_by_placeholder`.
2. Fill required fields and submit.
3. Wait for success toast and assert text.

Project C — Handle alert pop-up & assert text

1. Trigger action that raises `confirm("...")`.
2. Use `with page.expect_dialog() as dialog_info: page.click(...)`
3. Assert `dialog_info.value.message` and accept/dismiss accordingly.

Project D — Capture screenshot & trace after successful run

1. For screenshot:

   ```python
   page.screenshot(path="screenshots/test_todo.png", full_page=True)
   ```

2. For trace (requires Playwright tracing):

   ```python
   context.tracing.start(screenshots=True, snapshots=True)
   # run steps...
   context.tracing.stop(path="trace.zip")
   ```

3. Attach artifacts in CI.

Combined test template

```python
def test_todo_flow(page):
    page.goto("https://demo.playwright.dev/todomvc")
    page.locator("input.new-todo").fill("Buy milk")
    page.locator("input.new-todo").press("Enter")
    expect(page.locator("ul.todo-list li")).to_have_count(1)
    page.screenshot(path="screenshots/todo.png")
```

20 Interview Qs (Day 5)

1. **Q: How to capture full page screenshot?**
   A: `page.screenshot(path="x.png", full_page=True)`.
2. **Q: What is Playwright trace and why use it?**
   A: A trace contains screenshots, DOM snapshots, and network events for debugging failing tests.
3. **Q: How to start tracing?**
   A: `context.tracing.start()` then `context.tracing.stop(path="trace.zip")`.
4. **Q: Where to store screenshots/traces for CI?**
   A: Save as test artifacts in CI pipeline (GitHub Actions/others).
5. **Q: How to write end-to-end test that's reliable?**
   A: Stable locators, explicit waits, isolated state, mocking network if necessary, small tests.
6. **Q: How to run Playwright tests in headful mode to debug?**
   A: Launch browser with `headless=False` or use CLI flags.
7. **Q: How to parameterize tests in pytest?**
   A: Use `@pytest.mark.parametrize()` to run test variations.
8. **Q: How to assert that screenshot matches baseline?**
   A: Use image comparison tools — Playwright test runner has snapshot features; in pytest use image-diff libs.
9. **Q: How to reduce test run time while keeping coverage?**
   A: Parallelize, avoid redundant flows, run critical smoke tests more often.
10. **Q: How to design tests to be maintainable?**
    A: Page Object Model (POM)/helpers, small focused tests, use stable test IDs.
11. **Q: How to run only a single pytest test?**
    A: `pytest tests/test_file.py::test_name -q`.
12. **Q: How to capture network logs or HAR?**
    A: Intercept requests via `page.on("request")` and record, or use browser devtools protocol via other tools.
13. **Q: How to clean test data between runs?**
    A: Use APIs to reset state or create fresh contexts with `storage_state` or dedicated test accounts.
14. **Q: How to test file downloads?**
    A: Use `with page.expect_download() as d: page.click("download")` then `d.value.path()`.
15. **Q: How to attach artifacts in pytest output?**
    A: Use pytest hooks/plugins (allure, pytest-html) to attach screenshots and logs.

16. **Q: When should you mock network instead of using real backend?**
    A: For deterministic tests, when backend is flaky or slow, or to test edge cases.
17. **Q: How to test responsiveness / different viewports?**
    A: `context.new_page(viewport={"width":400,"height":800})` or `page.set_viewport_size()`.
18. **Q: What is flaky test detection strategy?**
    A: Re-run failures, collect flake data, root-cause by instrumenting logs/traces.
19. **Q: How to integrate Playwright tests into CI?**
    A: Install browsers in CI, run `pytest` or playwright test runner, upload artifacts on failure.
20. **Q: How to assert text inside a toast notification that disappears?**
    A: Use `expect(locator).to_be_visible()` then `to_have_text()` with proper timeout; capture before it disappears.

Quick tips & Suggested daily rhythm

- Start each day by running one passing test, then add one new assertion/locator/flow.
- Keep tests small: one behavior per test.
- Use `data-test-id` attributes in dev pages when possible; if you control app, add them.
- Use `expect()` for assertions because it retries and reduces flakiness.
- Always capture at least one screenshot or trace for failing CI runs.

If you want, I can:

- convert this into a **4-week roadmap** with milestones and Stage 6 + suggested enhancements (you previously asked about that) and produce a GitHub README template and a short repo description — I can generate all in one go. Want me to do that next?