SESIÓN 4 · QA ENGINEERING

Playwright Pro 🚀

Tu suite del Dojo Lab funciona... pero repite código y vive desordenada.
Hoy la conviertes en una **suite profesional**: hooks, POM real, fixtures y API testing.

`describe + hooks` `locators pro` `POM real` `fixtures` `API testing` `6 misiones 🎯`

DIAGNÓSTICO

Tu suite funciona. Ahora mírala con ojos de senior.

🔁

Síntoma 1: login repetido

4 de tus 5 specs empiezan con los mismos pasos de login. Copy-paste detectado.

Síntoma 2: selectores regados

El locator del botón de login vive en 4 archivos. Cambio de UI = cacería.

🐌

Síntoma 3: todo por UI

Cada verificación pasa por el navegador — lento y frágil para cosas que la API responde en milisegundos.

El plan de hoy

Acto 1 → **organiza**: describe + hooks

Acto 2 → **blinda**: locators semánticos

Acto 3 → **encapsula**: tu LoginPage real

Acto 4 → **inyecta**: fixtures propios

Acto 5 → **acelera**: API testing

Acto 6 → 🏆 suite reorganizada

🧬 Cada herramienta de hoy usa conceptos que YA dominas: hooks = funciones, POM = tu clase de Coding Classes, fixtures = el destructuring de `({ page })`.

test.describe: capítulos para tus tests 📂

```
test.describe("Login", () ⇒ {

  test("con credenciales válidas entra al dashboard", async ({ page }) ⇒ {
    // ...
  });

  test("con password inválida muestra error", async ({ page }) ⇒ {
    // ...
  });
});
```

📖 Agrupa por feature

El reporte se lee como índice: **Login →** sus casos. Con 100 tests, los capítulos salvan vidas.

🎯 Corre un capítulo

`npx playwright test -g "Login"` ejecuta solo ese grupo.

🔧 Herramientas de foco

`test.only(...)` corre SOLO ese test (debug) · `test.skip(...)` lo salta documentadamente. ⚠️ Nunca commitees un .only.

beforeEach: el login se escribe UNA vez 🪝

❌ ANTES — LOGIN COPIADO EN CADA TEST

```
test("ver dashboard", async ({ page }) => {
  await page.goto("/login");
  await page.getByLabel("Email").fill("testuser@upex.dev");
  await page.getByLabel("Password").fill("Test123!");
  await page.getByRole("button", { name: /sign in/i }).click();
  // ...por fin el test de verdad
});
// ...y otra vez en el siguiente test 😩
```

✅ DESPUÉS — EL HOOK LO HACE POR TODOS

```
test.describe("Dashboard", () => {

  test.beforeEach(async ({ page }) => {
    await page.goto("/login");
    // ...pasos de login, UNA sola vez
  });

  test("ver dashboard", async ({ page }) => {
    // arranca YA logueado 🎉
  });

  test("crear tarea", async ({ page }) => {
    // también logueado
  });
});
```

🪝 **beforeEach corre antes de CADA test del grupo** — cada test arranca con página fresca y logueada, independiente de los demás. También existen `afterEach`, `beforeAll`, `afterAll` (limpieza y setup pesado).

MISIÓN P1

Refactoriza con describe + beforeEach

☐ Completada

🎯 Reorganizar tu suite del lab: agrupa los tests por feature y mueve el login repetido a un **beforeEach**.

1. Agrupa tus specs con `test.describe` por feature
2. Donde 2+ tests repitan login → `test.beforeEach`
3. Corre la suite completa de nuevo:

```
$ npx playwright test
```

💡 **Pista: un test ahora falla**

🪚 Métrica de éxito honesta: tu suite debería perder ~15-25 líneas duplicadas sin perder ningún test.

✅ **Criterio de éxito:** misma cantidad de tests, todos verdes, login escrito UNA sola vez por grupo.

⭐ QUIZ 1 · Hooks

Un describe tiene 3 tests y un `beforeEach` con login. ¿Cuántas veces se ejecuta el login al correr el grupo?

A 1 vez — al inicio del grupo

B 3 veces — antes de CADA test

C Depende del orden de los tests

💡 before**Each** = antes de CADA test. Eso garantiza **aislamiento**: cada test arranca limpio, sin heredar el estado (ni el desorden) del anterior. La opción A describe a `beforeAll` — más rápido pero comparte estado: el trade-off clásico velocidad vs aislamiento.

La jerarquía: elige locators que sobreviven 🛡️

PRIORIDAD	LOCATOR	POR QUÉ
🥇 1°	`getByRole("button", { name: "Sign in" })`	Cómo lo percibe un usuario (y un lector de pantalla). Sobrevive rediseños.
🥈 2°	`getByLabel("Email")` · `getByPlaceholder(...)`	Anclado al texto visible del formulario.
🥉 3°	`getByText("Bienvenida")`	Para textos y mensajes no interactivos.
4°	`getByTestId("submit-btn")`	Ancla dedicada a testing (`data-testid`) — pídela al equipo dev.
🧨 último	`page.locator(".btn > div:nth-child(2)")`	CSS estructural: muere con cualquier rediseño. Solo sin alternativa.

♿ **Bonus oculto:** si getByRole no encuentra tu botón, a un usuario con lector de pantalla le pasa lo mismo. Locators semánticos = tests robustos Y auditoría de accesibilidad gratis.

Encadenar y filtrar: apuntar fino 🎯

El problema: hay 10 filas, quiero UNA

```
// "el botón Delete de la fila de Ana"
await page
  .getByRole("row")
  .filter({ hasText: "Ana" })
  .getByRole("button", { name: "Delete" })
  .click();
```

Se lee como la frase: fila → que contenga "Ana" → su botón Delete.
Encadenar = acotar el contexto paso a paso.

Aserciones web-first (esperan solas)

```
await expect(locator).toBeVisible();
await expect(locator).toHaveText("Guardado");
await expect(locator).toHaveCount(3);
await expect(locator).toBeEnabled();
await expect(page).toHaveURL(/dashboard/);
```

Reintentan automáticamente hasta cumplirse (o timeout). **Por eso nunca necesitas sleep**: la aserción ES la espera.

MISIÓN P2

Caza de locators frágiles

☐ Completada

🎯 Auditar tu suite y reemplazar todo selector CSS estructural por locators **semánticos** (getByRole / getByLabel / getByText).

1. Busca en tus specs: `page.locator("` con CSS
2. Por cada uno, encuentra el equivalente semántico — el **locator picker del UI mode** te lo sugiere:

```
$ npx playwright test --ui
```

3. Reemplaza, corre, verde.

💡 **Pista: el dojo no tiene label en un input**

🧪 Prueba ácida: ¿tu locator describe QUÉ es el elemento ("el botón Sign in") o DÓNDE está colgado ("el tercer div del form")? Solo el primero sobrevive.

✅ **Criterio de éxito:** cero selectores CSS estructurales en tu suite (o cada excepción justificada con comentario) y todo verde.

⭐ QUIZ 2 · Locators

El equipo rediseña el CSS completo de la app (clases nuevas, estructura nueva, mismos textos). ¿Cuál locator sigue funcionando?

A `page.locator(".btn-primary")`

B `page.locator("form > div:nth-child(3) > button")`

C `page.getByRole("button", { name: "Sign in" })`

💡 El rediseño cambió clases y estructura (matando A y B) pero el botón sigue SIENDO un botón que dice "Sign in" — el rol y el nombre accesible son el contrato con el usuario, y ese contrato es lo último que cambia. Locator semántico = apostarle al contrato, no a la implementación.

Tu LoginPage... ahora de verdad 📄

pages/login-page.ts

```ts
import { type Page, type Locator } from "@playwright/test";

export class LoginPage {
  readonly emailInput: Locator;
  readonly passwordInput: Locator;
  readonly signInButton: Locator;

  constructor(private page: Page) {
    this.emailInput = page.getByLabel("Email");
    this.passwordInput = page.getByLabel("Password");
    this.signInButton = page.getByRole("button", { name: /sign in/i });
  }

  async goto() {
    await this.page.goto("https://dojo.upexgalaxy.com/login");
  }

  async login(email: string, password: string) {
    await this.emailInput.fill(email);
    await this.passwordInput.fill(password);
    await this.signInButton.click();
  }
}
```

🆕 vs Coding Classes

Misma clase que construiste en el simulador — con 3 upgrades de producción.

📍 Locators como campos

Declarados UNA vez en el constructor, tipados como `Locator`, reutilizados en todos los métodos.

🏷️ TypeScript real

`type Page` importado, `readonly` (no reasignable), `private page` = parámetro y propiedad en un paso.

📤 export

La palabra que permite a otros archivos hacer `import { LoginPage }`.

El spec se queda limpio y legible

tests/login.spec.ts

```ts
import { test, expect } from "@playwright/test";
import { LoginPage } from "../pages/login-page";

test.describe("Login", () => {
  let loginPage: LoginPage;

  test.beforeEach(async ({ page }) => {
    loginPage = new LoginPage(page);
    await loginPage.goto();
  });

  test("credenciales válidas → dashboard", async ({ page }) => {
    await loginPage.login("testuser@upex.dev", "Test123!");
    await expect(page).toHaveURL(/dashboard/);
  });
});
```

🗂️ Estructura nueva

```
mi-dojo-lab/
├─ pages/    ← tus Page Objects
│  └─ login-page.ts
└─ tests/    ← solo specs
   └─ login.spec.ts
```

📜 La división sagrada

Acciones → viven en el Page Object.
Aserciones (expect) → viven en el test.
El test decide qué es pasar/fallar; la página solo sabe operarse.

MISIÓN P3

Tu primer Page Object de producción

Completada

🎯 Crear **pages/login-page.ts** y refactorizar tus specs de login para usarlo.

1. Crea la carpeta `pages/` y el archivo `login-page.ts` (modelo en slide 11 — tipéalo, no lo pegues: la memoria muscular importa)
2. Refactoriza login.spec.ts y login-invalid.spec.ts para usarlo
3. Verde de nuevo:

```
$ npx playwright test login
```

💡 **Pista: "Cannot find module '../pages/login-page'"**

💡 **Reto extra si vas sobrado**

✅ **Criterio de éxito:** tests de login verdes SIN ningún `page.fill/page.click` directo en los specs — todo pasa por LoginPage.

★ QUIZ 3 · POM

Según la división sagrada, ¿dónde vive `expect(page).toHaveURL(/dashboard/)`?

A En LoginPage — al final del método login()

B En el test — el Page Object actúa, el test verifica

C En playwright.config.ts

💡 Si login() incluyera el expect del dashboard, no podrías reusarlo para el caso "login inválido se queda en /login" — el método fallaría por diseño. Las acciones van al POM porque se REPITEN; las aserciones van al test porque CAMBIAN según el caso. Reusabilidad vs especificidad.

Fixtures: crea tu propio ({ page }) 💉

fixtures.ts

```ts
import { test as base } from "@playwright/test";
import { LoginPage } from "./pages/login-page";

export const test = base.extend<{ loginPage: LoginPage }>({
  loginPage: async ({ page }, use) => {
    const loginPage = new LoginPage(page);
    await loginPage.goto();
    await use(loginPage);  // ← aquí corre el test
  },
});
export { expect } from "@playwright/test";
```

tests/login.spec.ts — ahora

```ts
import { test, expect } from "../fixtures";

test("login válido", async ({ loginPage, page }) => {
  await loginPage.login("testuser@upex.dev", "Test123!");
  await expect(page).toHaveURL(/dashboard/);
});
```

🤯 El círculo se cierra

`({ page })` que usas desde Coding Base ES un fixture de Playwright. Acabas de aprender a fabricar los tuyos: `({ loginPage })` llega **ya instanciada y en la página de login** — sin new, sin goto, sin beforeEach.

MISIÓN P4

Fabrica tu fixture

☐ Completada

🎯 Crear **fixtures.ts** con el fixture `loginPage` y migrar tus specs de login a usarlo.

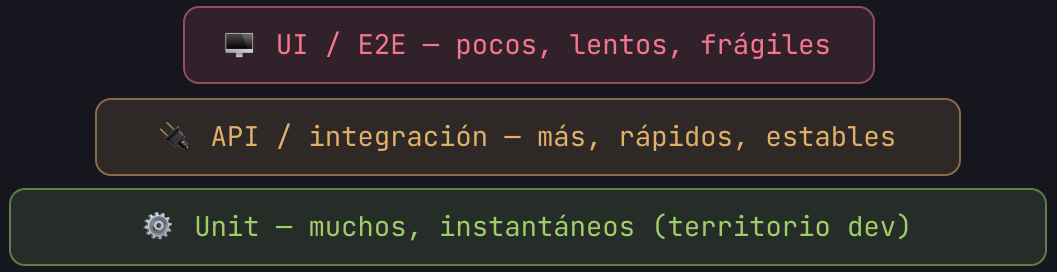
1. Crea `fixtures.ts` en la raíz (modelo en slide 15)
2. En tus specs de login: cambia el import de `@playwright/test` por `../fixtures`
3. Recibe `({ loginPage, page })` y borra el beforeEach que ya no necesitas
4. Verde de nuevo.

💡 **Pista: TypeScript se queja del extend**

🥋 Sembrando futuro: en KATA recibirás `({ api })`, `({ ui })`, `({ test })` — fixtures exactamente como el que acabas de fabricar, con más capas dentro.

✅ **Criterio de éxito:** tus tests de login reciben `({ loginPage })` del import de fixtures propio y pasan en verde.

No todo merece un navegador: la pirámide 🔺



Tu test de login por UI: **~5 segundos** (browser, render, animaciones). El mismo login por API: **~200 ms** — y no se rompe cuando muevan un botón.

🖥️ UI cuando...

Pruebas el VIAJE del usuario: flujos críticos, visual, interacción real.

🔌 API cuando...

Pruebas REGLAS de negocio: validaciones, permisos, datos, errores 4xx. La mayoría de tus casos edge.

🥋 El dojo te espera

Tiene API REST documentada: `dojo.upexgalaxy.com/api/docs` — ábrela AHORA y mira qué endpoints ofrece.

request: Playwright sin navegador 🔌

```ts
test("API: login devuelve sesión", async ({ request }) => {
  const res = await request.post("https://dojo.upexgalaxy.com/api/auth/login", {
    data: { email: "testuser@upex.dev", password: "Test123!" },
  });

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

  const body = await res.json();
  expect(body).toHaveProperty("user");
});

test("API: login inválido responde 401", async ({ request }) => {
  const res = await request.post(".../api/auth/login", {
    data: { email: "testuser@upex.dev", password: "mala" },
  });
  expect(res.status()).toBe(401);
});
```

🗡️ ({ request })

Otro fixture de la casa — cliente HTTP listo, cero navegador.

📨 Anatomía

`post(url, { data })` envía JSON · `res.status()` el código · `res.json()` el cuerpo. Los status (200, 401, 404…) ya los conoces del oficio.

🗺️ Forma vs realidad

Rutas y campos exactos: en `/api/docs` del dojo. Estas slides muestran la FORMA — regla de oro del lab.

MISIÓN P5

Tus primeros API tests

☐ Completada

🎯 Explorar **/api/docs** del dojo y escribir 2 tests de API en `tests/api/`.

1. Abre `dojo.upexgalaxy.com/api/docs` y elige 2 endpoints
2. Sugerencia: login OK (200) y login inválido (401)
3. Créalos en `tests/api/auth.spec.ts` y corre solo esa carpeta:

```
$ npx playwright test tests/api
```

💡 **Pista: ¿qué ruta y campos exactos?**

⚡ Fíjate en la duración al terminar: tus API tests deberían correr en **menos de 2 segundos**. Compáralo con tus tests de UI — ese delta ES la pirámide.

✅ **Criterio de éxito:** 2 API tests verdes en `tests/api/`, validando status y al menos una propiedad del body.

★ QUIZ 4 · Estrategia

Debes probar 12 combinaciones de datos inválidos en el formulario de registro. ¿Estrategia profesional?

A 12 tests de UI — el formulario es UI, se prueba por UI

B 12 por API (las reglas de validación) + 1-2 por UI (que el error se MUESTRE al usuario)

C Probar solo las 3 más probables y rezar

💡 Las REGLAS de validación viven en el backend → API: 12 casos en ~2 segundos, estables. Lo que la UI aporta de único es MOSTRAR el error → 1-2 tests de UI lo cubren. División del trabajo: API prueba la regla, UI prueba la experiencia. Cobertura completa, suite rápida.

MISIÓN P6 · BOSS 🏆

La suite profesional, completa

☐ Completada

🎯 Todo junto: estructura final ordenada, suite completa verde, reporte como evidencia.

```
mi-dojo-lab/
├─ fixtures.ts        ← tus fixtures (P4)
├─ pages/
│  └─ login-page.ts   ← POM (P3)
├─ tests/
│  ├─ login.spec.ts   ← describe+fixture (P1, P4)
│  ├─ register.spec.ts
│  ├─ navigation.spec.ts
│  └─ api/
│     └─ auth.spec.ts   ← API tests (P5)
└─ playwright.config.ts
```

```
$ npx playwright test
```

```
$ npx playwright show-report
```

📊 En el reporte: tus describe como capítulos, API tests en milisegundos, UI tests organizados. **Esta estructura es una mini-versión de cómo se ve un framework profesional.**

💡 **Pista: specs que aún usan page.fill para login**

✅ **Criterio de éxito:** árbol como el de la izquierda, suite completa verde (UI + API), reporte HTML navegable por capítulos.

LO QUE ACABAS DE CONSTRUIR

Tu carpeta ya rima con KATA 🥋

LO TUYO (HOY)	KATA (EL FRAMEWORK PROFESIONAL)
`pages/login-page.ts`	Page Component (capa L3) — con ATCs registrados
`fixtures.ts` con `({ loginPage })`	TestFixture (L4) — entrega `({ ui })`, `({ api })`, `({ test })`
Helpers repetidos en cada POM	UiBase / ApiBase (L2) — heredados con `extends`
Credenciales hardcodeadas 😬	TestContext (L1) — config + datos por ambiente
`tests/api/auth.spec.ts`	Api Components con tipos generados del OpenAPI

 No aprendiste "Playwright avanzado" — construiste, a mano y a escala chica, **cada pieza que KATA industrializa**. El próximo deck cruza el puente.

DEBRIEF

Misiones completadas

Tarea

Termina misiones pendientes + crea el POM de una segunda página del dojo.

Siguiente: Dev Craft

Git, GitHub y tu primer PR — para que esta suite viva en equipo, no en tu laptop.

Después: KATA Bridge

El último deck: cruzar de tu mini-framework al boilerplate profesional.