

QA ENGINEERING · PATRONES DE AUTOMATIZACIÓN

Automation Patterns

De clases sueltas a una **arquitectura de pruebas**.

Page Object · Fixtures · Paralelización · KATA — los patrones que sostienen todo framework profesional.

POM

Fixtures (DI)

Parallel workers

KATA

código real Playwright

6 quizzes ★

Ya escribes tests. Hoy los conviertes en arquitectura.

✓ Lo que ya tienes

JS moderno · clases + `extends` · un Page Object que envuelve una página · tu primer `test()` ejecutado.

🎯 Lo que sumas hoy

Los **4 patrones** que escalan de 5 a 5.000 tests sin que el proyecto colapse: **POM** a fondo · **Fixtures** · **Paralelización** · **KATA**.

*// Un patrón = una respuesta probada
// a un dolor que SIEMPRE reaparece:*

selector cambió → *Page Object*

setup repetido → *Fixtures*

suite lentísima → *Paralelización*

todo junto, a escala → *KATA*



Cada acto: **primero el dolor**, luego el patrón que lo cura.
Nunca sintaxis sin motivo.

★ QUIZ relámpago • ¿Caliente de la sesión anterior?

Un Page Object es, en una frase...

- ☐ A Un archivo de configuración de Playwright
- ☒ B Una `c\l`ass que envuelve una página y guarda sus selectores + acciones en un solo lugar
- ☐ C Una función que abre el navegador

💡 Page Object Model (POM): una `c\l`ass donde viven los selectores y las acciones de UNA página. Los tests la usan sin tocar selectores. Hoy lo llevamos más lejos — y descubrimos que es solo el **primero** de cuatro patrones que se apilan.

El dolor: selectores regados por todo el spec 🔍

login.spec.ts – sin patrón

```
test("login exitoso", async ({ page }) => {  
  await page.fill("#email", "ana@qa.com");  
  await page.fill("#password", "secreto123");  
  await page.click("#login-btn");  
  await expect(page.locator("#welcome")).toBeVisible();  
});  
  
test("login inválido", async ({ page }) => {  
  await page.fill("#email", "ana@qa.com");  
  await page.fill("#password", "malamala");  
  await page.click("#login-btn");  
  await expect(page.locator("#error-msg")).toBeVisible();  
});
```

*mismos selectores,
copiados en cada test*

... otra vez ...



Pregunta asesina: si #login-btn cambia de id, ¿cuántos lugares corriges? Con 50 tests → **50 ediciones** y un riesgo de error en cada una.

page object

fixtures

paralelización

kata

El patrón: una clase = una página 📁

```
class LoginPage {
  constructor(page) {
    this.page = page;          // el "control remoto"
  }

  // selectores: una sola fuente
  email    = () => this.page.locator("#email");
  password = () => this.page.locator("#password");
  submit   = () => this.page.locator("#login-btn");

  async login(email, password) {
    await this.email().fill(email);
    await this.password().fill(password);
    await this.submit().click();
  }
}
```

📌 Selectores arriba, una vez

Cada selector vive en UN getter. ¿Cambió `#login-btn` ?
Editas **1 línea** — los 50 tests ni se enteran.

🎬 Acciones como métodos

`login()` cuenta una historia: llena, llena, clic. El test lee como un caso manual.

🧬 Cero conceptos nuevos

Es la clase de siempre: `constructor` · `this` · método
`async` · `await` . POM = clase + propósito.

page object

fixtures

paralelización

kata

La regla de oro: acciones dentro, decisión fuera

El Page Object: solo ACCIONES

```
class LoginPage {  
  async login(email, pass) {  
    await this.email().fill(email);  
    await this.password().fill(pass);  
    await this.submit().click();  
  }  
} // hace cosas, no juzga
```

El test: la DECISIÓN pasa/falla

```
test("login exitoso", async ({ page }) => {  
  const login = new LoginPage(page);  
  await login.login("ana@qa.com", "secreto123");  
  
  await expect(page.locator("#welcome"))  
    .toBeVisible(); // aquí se decide  
});
```



Misma acción `login()` sirve para el caso feliz y el inválido — solo cambia el `expect` que pone el test. Reutilización máxima, decisión en un solo lugar visible. **Esta división reaparece idéntica en KATA.**

Tu Page Object, controlando el navegador 🤖

https://bunkai-bank.test/login

 **Bunkai Bank**

Acceso a tu cuenta

EMAIL

PASSWORD

Entrar

login-page.spec.ts – clase real, ejecución real

⚡ Ejecutable en la versión online

```
class LoginPage {
  constructor(page) {
    this.page = page;
  }
  async login(email, password) {
    await this.page.fill("#email", email);
    await this.page.fill("#password", password);
    await this.page.click("#login-btn");
  }
}

test("login con Page Object", async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.login("ana@qa.com", "secreto123");
  await expect(page.locator("#welcome")).toBeVisible();
});
```

► Ejecuta – tu CLASE va a mover el cursor...

★ QUIZ 1 • Page Object

El dev renombra `#login-btn` → `#signin-btn`. Tienes 50 tests que hacen login con `LoginPage`. ¿Cuántos lugares corriges?

- ☐ A 50 — uno por test
- ☐ B Ninguno — Playwright lo detecta solo
- ☒ C 1 — el selector vive solo dentro de `LoginPage`

💡 Esa es TODA la promesa del Page Object: los selectores viven en un único lugar. Los 50 tests llaman `login.login(...)` y no se enteran del cambio. La opción B es el optimista mágico — ninguna herramienta adivina un renombre.

page object

fixtures

paralelización

kata

Nuevo dolor: cada test repite su preparación

```
test("ve su dashboard", async ({ page }) => {  
  const login = new LoginPage(page);  
  await login.goto();  
  await login.login(EMAIL, PASS);  
  const dash = new DashboardPage(page);  
  await dash.expectVisible();  
});
```

```
test("edita su perfil", async ({ page }) => {  
  const login = new LoginPage(page);  
  await login.goto();  
  await login.login(EMAIL, PASS);  
  // y recién aquí empieza el test real  
});
```

🧩 El síntoma

Construir objetos, navegar, autenticar — **antes** de cada test. Ruido que tapa lo que el test realmente prueba.

🧹 ¿Y la limpieza?

¿Quién cierra sesión, borra el usuario creado, resetea el estado **después**? Hoy: nadie, o copiado en cada test.

💡 La idea

Que alguien prepare el escenario **antes** y lo limpie **después**, y le entregue al test solo lo que pidió, listo para usar. Eso es una **Fixture**.

page object

fixtures

paralelización

kata

Sorpresa: llevas toda la vida usando fixtures 🎭

```
test("login", async ({ page }) => {  
  //           ▲  
  //           esto YA es una fixture  
  await page.goto("/login");  
});
```



Playwright construye un page fresco **antes** de tu test, te lo presta, y lo **desecha después**. Tú nunca escribiste `new Page()` ni lo cerraste. Eso es una fixture trabajando.



¿Qué es una fixture?

Un **recurso listo para usar** que el framework arma, te inyecta por nombre, y limpia solo. Inyección de Dependencias (DI) aplicada a tests.



Built-in de Playwright

`page` · `request` · `context` · `browser` . Pides por nombre lo que usas; lo demás ni se crea.



Lo nuevo de hoy

Crear las **tuyas** con `test.extend` : una fixture `LoginPage` , una `authedUser` ... que lleguen ya preparadas a cada test.

page object

fixtures

paralelización

kata

Fabricar una fixture: setup → use → teardown 🛠️

```
import { test as base } from "@playwright/test";

export const test = base.extend({
  // fixture llamada "loginPage"
  loginPage: async ({ page }, use) => {
    // 1 · SETUP (antes del test)
    const login = new LoginPage(page);
    await login.goto();

    // 2 · entregar al test
    await use(login);

    // 3 · TEARDOWN (después del test)
    await page.close();
  },
});
```

1 · Antes de use

Todo lo que escribes **arriba** de `use()` corre como preparación, una vez por test que la pida.

2 · await use(value)

La frontera. `value` es lo que el test recibe en `({ loginPage })`. El test corre **durante** esta línea.

3 · Después de use

Todo lo de **abajo** es limpieza: cerrar, borrar, resetear. Corre **siempre**, incluso si el test falla.

Qué pasa en `await use(...)`, paso a paso

```
loginPage: async ({ page }, use) => {  
  const login = new LoginPage(page);  
  await login.goto();  
  await use(login); // ← frontera  
  await page.close();  
}  
  
test("dashboard", async ({ loginPage }) => {  
  await loginPage.login(EMAIL, PASS);  
  await expect(dash).toBeVisible();  
});
```

▶ Siguiente paso

↺ Reiniciar

paso 0 / 6

📌 ¿DÓNDE CORRE EL CONTROL?

— en reposo —

Presiona "Siguiente paso": vas a ver el control saltar de la fixture al test y volver.

consola...

Dos superpoderes: perezosas y combinables ⚡

😴 Lazy: solo se crea lo que pides

```
// pide solo "api" → NO abre navegador
test("GET /orders", async ({ api }) => {
  await api.orders.getOrders({ limit: 10 });
});

// pide "ui" → SÍ abre navegador
test("ve la lista", async ({ ui }) => {
  await ui.orders.navigateTo();
});
```

Playwright instancia **solo** las fixtures que el test nombra. Tests de API puros no pagan el costo de un navegador.

🧩 Combinables: fixtures sobre fixtures

```
base.extend({
  // authedUser DEPENDE de api
  authedUser: async ({ api }, use) => {
    const user = await api.users.create();
    await api.auth.login(user);
    await use(user);
    await api.users.remove(user.id); // limpia
  },
});
```

Una fixture pide otras en su `({ ... })`. Playwright resuelve la cadena y el orden de limpieza por ti.



Estos dos poderes son el motor de KATA: `{ api }` sin navegador, `{ ui }` con navegador, `{ test }` combinando ambos — fixtures que dependen de fixtures.

★ QUIZ 2 · Fixtures

En una fixture, ¿qué hace exactamente `await use(value)`?

☐ A Importa el valor desde otro archivo

☒ B Entrega `value` al test y pausa la fixture; al terminar el test, sigue con el teardown de abajo

☐ C Verifica que el valor no sea `undefined`

💡 `use()` parte la fixture en dos: lo de **arriba** es setup, lo de **abajo** es teardown, y el **test corre en medio**. Por eso la limpieza ocurre siempre al final — incluso si el test falla. Es el sándwich `setup → use → teardown`.

page object

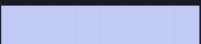
fixtures

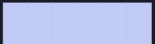
paralelización

kata

Nuevo dolor: 300 tests, uno tras otro, 40 minutos

// Secuencial: la suma de TODO

test 1  8s

test 2  6s

test 3  ...

 total = 8+6+...+N
= la suite completa, sumada



En CI, una suite secuencial de 300 tests bloquea cada PR media hora. El feedback lento mata el shift-left.

El síntoma

Tu máquina tiene 8 núcleos y usas **1**. Los otros 7 miran cómo un solo test corre a la vez.

La idea

Repartir los tests entre varios **workers** (procesos) que corren **a la vez**. El reloj de pared baja al del worker más cargado, no a la suma.

El requisito

Para correr en paralelo, los tests deben ser **independientes**: sin estado compartido, sin depender del orden.

page object

fixtures

paralelización

kata

El patrón: workers = procesos aislados 🧑‍🔧

🧑‍🔧 Un worker = un proceso

Playwright lanza varios **procesos de Node**. Cada uno toma tests del lote y los corre. Por defecto reparte tests a `~½ de tus núcleos`.

🧱 Aislamiento total

Cada worker tiene su **propio navegador** y sus **propias fixtures**. Lo que pasa en un worker no toca a otro — por eso la independencia importa.

📁 Por defecto: archivos en paralelo

Playwright corre **archivos** distintos en paralelo; los tests **dentro** de un archivo, en orden. `fullyParallel: true` paraleliza también dentro del archivo.

```
// playwright.config.ts
export default defineConfig({
  // n° de procesos en paralelo
  workers: process.env.CI ? 4 : undefined,

  // paraleliza tests DENTRO de cada archivo
  fullyParallel: true,
});

// control fino por bloque:
test.describe.configure({ mode: "parallel" });
test.describe.configure({ mode: "serial" });
```

📄 workers = cuántos procesos. fullyParallel = si los tests de un mismo archivo también se reparten.

🎮 SIMULADOR · MUEVE LOS WORKERS, MIRA EL RELOJ

Reparte 6 archivos entre 1, 2 o 4 workers

workers:

1

2

4

⚡ Ejecutable en la versión online

👤 worker 1

checkout.spec · 4s

cart.spec · 2s

orders.spec · 2s

👤 worker 2

login.spec · 3s

search.spec · 3s

profile.spec · 1s

RELOJ DE PARED

8.0s

TRABAJO TOTAL (CPU)

15.0s

ACELERACIÓN VS 1 WORKER

1.88×

page object

fixtures

paralelización

kata

Más allá: sharding · y por qué este repo va con cuidado

Sharding: repartir entre máquinas

```
# máquina 1 de 3 en CI
npx playwright test --shard=1/3
# máquina 2 de 3
npx playwright test --shard=2/3
# máquina 3 de 3
npx playwright test --shard=3/3
```

Workers = paralelo dentro de una máquina. **Shards** = paralelo entre **varias máquinas** de CI, y luego se fusionan los reportes.

La postura honesta de este repo

```
// playwright.config.ts (real)
fullyParallel: false,
retries: 0,
// "Single worker for now - increase
// when tests are stable"
workers: 1,
```



Primero **determinismo**, después velocidad. Con la suite joven se corre en serie para cazar dependencias ocultas; el paralelismo se sube cuando los tests ya son sólidos.

★ QUIZ 3 · Paralelización

Subes de 1 a 4 workers y la suite pasa de 40 a 12 min. ¿Qué bajó?

- ☐ A El trabajo total de CPU — se ejecuta menos código
- ☒ B El **reloj de pared** — el mismo trabajo se reparte entre 4 procesos a la vez
- ☐ C El número de tests — Playwright salta los repetidos

💡 El cómputo total es idéntico: se ejecutan los mismos tests, el mismo código. Lo que cambia es el **tiempo de pared**, porque 4 workers trabajan en paralelo. Y el límite lo pone el worker más cargado — por eso 4 workers casi nunca dan exactamente 4×.

page object

fixtures

paralelización

kata

KATA: los tres patrones, ordenados en capas

KATA (**K**omponent **A**ction **T**est **A**rchitecture) no inventa nada nuevo: **organiza** lo que ya viste en tres capas nombradas sobre una base de utilities y fixtures, donde cada capa `extends` la anterior.

POM → capa de Componentes

Tu `LoginPage` es un componente de dominio. Ahí viven las **ATCs**.

Fixtures → capa 4

`{ api }`, `{ ui }`, `{ test }` inyectan los componentes ya listos.

Paralelización → config

La misma `playwright.config.ts`: workers, projects, dependencias.

// La cadena de herencia REAL del repo

```
class TestContext { ... }           // L1
class UiBase extends TestContext    // L2
class LoginPage extends UiBase      // L3 ← tu POM
```

// y la fixture los junta:

```
class UiFixture {
  login = new LoginPage(options);
}
```



El `extends` que dominaste es el esqueleto. KATA es Page Objects + Fixtures con capas y nombres.

page object

fixtures

paralelización

kata

Las capas de KATA 🏛️

L4 · Fixtures (DI)

TestFixture · ApiFixture · UiFixture — inyectan todo

↑ usa

L3.5 · Steps

cadenas de ATCs reutilizables (precondiciones)

↑ usa

★ L3 · Componentes de dominio

LoginPage · UsersApi — aquí viven las ATCs

↑ extends

L2 · Bases

ApiBase (HTTP) · UiBase (Playwright)

↑ extends

L1 · TestContext

config · logger · faker · entorno



Regla única: una capa superior puede usar una inferior, **nunca** al revés. Tu Page Object (L3, estrella) es el corazón — todo lo demás existe para servirlo o consumirlo.

page object

fixtures

paralelización

kata

La pieza nueva: la ATC

```
class LoginPage extends UiBase {  
  
    @atc("PROJ-101")  
    async loginSuccessfully(creds) {  
        await this.fillAndSubmit(creds);  
        await this.page.waitForURL(  
            url => !url.pathname.includes("/login"));  
        await expect(this.page)  
            .not.toHaveURL(/.*\/login.*/);  
    }  
}
```

ATC = un caso de prueba completo

Acceptance Test Case: no un clic suelto, sino un **mini-flujo** entero (precondición → acción → aserción fija).

@atc("PROJ-101")

El decorador ata la ATC a su **ticket** de Jira 1:1. Da trazabilidad y tracing automático. Es un método con etiqueta.

Atómica

Una ATC **nunca** llama a otra ATC. ¿Cadenas reutilizables? Eso es un **Step** (capa 3.5).

Un test llama `ui.login.loginSuccessfully()` — ¿por dónde viaja?

```
test("PROJ-101: login", async ({ ui }) => {  
  await ui.login.goto();  
  await ui.login.loginSuccessfully(creds);  
});
```

```
// L4 UiFixture.login = new LoginPage(opts)  
// L3 class LoginPage extends UiBase  
// L2 class UiBase extends TestContext  
// L1 class TestContext { config, faker }
```

▶ Siguiente paso

↶ Reiniciar

paso 0 / 5

🏛️ CAPA ACTIVA

— en reposo —

Presiona "Siguiente paso": vas a seguir la llamada bajando por las capas de KATA.

page object

fixtures

paralelización

kata

En la práctica: pides la fixture según lo que pruebas

TIPO DE TEST	FIXTURE	¿NAVEGADOR?	CUÁNDO
API pura (integration)	<code>{ api }</code>	No	Testing de API. Default en <code>tests/integration/</code>
Solo UI	<code>{ ui }</code>	Sí	Flujo de UI sin setup por API
Híbrido	<code>{ test }</code>	Sí	Setup por API + acción UI + verificación API
Precondiciones repetidas	<code>{ steps }</code>	Depende	3+ ATCs repetidas en 3+ archivos

```
// API pura – NO abre navegador (lazy)
test("PROJ-7: lista pedidos", async ({ api }) => {
  await api.orders.getOrders({ limit: 10 });
});
```

```
// Híbrido – crea por API, verifica por UI
test("PROJ-9: aparece en lista", async ({ test }) => {
  const [, o] = await test.api.orders.create(data);
  await test.ui.orders.verifyVisible(o.id);
});
```

★ QUIZ 4 • KATA

Vas a probar solo un endpoint GET `/orders`, sin tocar UI. ¿Qué fixture pides?

A { `api` } — API pura, no abre navegador (lazy)

B { `ui` } — siempre conviene tener el navegador abierto

C { `test` } — para estar seguros, pide todo

💡 { `api` }: como las fixtures son **lazy**, pedir solo `api` NO arranca un navegador — el test corre en milisegundos. Pedir { `ui` } o { `test` } "por si acaso" desperdicia segundos de arranque de browser en cada test. Pide lo mínimo que uses.

★ QUIZ 5 · El integrador

Conecta cada dolor con el patrón que lo cura:

A selector regado → Fixtures · setup repetido → POM · suite lenta → KATA

B selector regado → **POM** · setup/teardown repetido → **Fixtures** · suite lenta → **Paralelización**

C todos los dolores → un solo patrón mágico

💡 Cada patrón ataca un dolor distinto: **POM** centraliza selectores, **Fixtures** ordenan el setup/teardown, **Paralelización** baja el reloj. Y **KATA** es el marco que los apila en capas para que escalen juntos. Cuatro herramientas, cuatro problemas.

Dónde vive cada patrón en este repo

```
tests/
  components/
    TestContext.ts           // L1
    ui/UiBase.ts            // L2
    ui/LoginPage.ts         // L3 · POM + ATCs
    api/ApiBase.ts          // L2
    steps/*.ts              // L3.5 · Steps
    TestFixture.ts         // L4 · Fixtures
    UiFixture.ts            // L4
  e2e/**                    // tests UI (+API)
  integration/**            // tests API puros
  playwright.config.ts      // workers, projects
```

Tu zona de trabajo

El 90% del tiempo: escribir **ATCs** en `components/ui/` y `components/api/`, y orquestarlas en `e2e/` · `integration/`.

Ya hecho por ti

TestContext, las Bases y las Fixtures vienen en el boilerplate. Heredas helpers; no los reescribes.

Para profundizar

Skill `/test-automation` + `references/kata-architecture.md` : reglas completas de ATC, fixtures y Steps.

Cuatro patrones, un mapa

PATRÓN	DOLOR QUE CURA	PIEZA CLAVE	EN KATA
 Page Object	selectores regados	<code>c\l ass</code> = una página	Componentes (L3)
 Fixtures	setup/teardown repetido	<code>test.extend + use()</code>	Fixtures (L4)
 Paralelización	suite lentísima	<code>workers</code> aislados	config + projects
 KATA	todo junto, a escala	arquitectura de capas + ATC	el marco completo



Cruzaste la frontera entre **escribir tests** y **diseñar una arquitectura de pruebas**. Lo mismo que hace un SDET senior, ahora con nombre y forma.

MISIÓN CUMPLIDA

Tu puntaje final

— / 6



Re-juega en casa

Este archivo corre offline: vuelve al simulador POM, al de workers y a los steppers. Toca todo.



Reto puente

Abre

`tests/components/ui/LoginPage.ts`

del repo y ubica: capa, ATC, decorador

`@atc`, aserciones fijas.



Próximo paso

Escribe tu primera ATC real con

`/test-automation` — Plan → Code →

Review sobre KATA.

Hoy: POM a fondo → Fixtures (`use()`) → Paralelización (workers) → **KATA**. De script a arquitectura.

QA ENGINEERING · AUTOMATION PATTERNS

De script a arquitectura

POM · Fixtures · Paralelización · KATA.

Cuatro patrones, una forma de pensar las pruebas.

[¿Preguntas?](#)

</test-automation>

<references/kata-architecture.md>