QA ENGINEERING · AUTOMATIZACIÓN DE PRUEBAS

API Automation con Playwright

Probar el **backend** directo — sin navegador, sin clics.
La fixture `request` · aserciones de respuesta · validación de schema · KATA para API.

request fixture | status + body | schema | KATA · ApiBase | runner real | 6 quizzes ⭐

EL SALTO DE HOY

Bajas un piso: del navegador a la API 🪜

✅ Lo que ya sabes

Escribir un `test()` · Page Objects · fixtures con `use()`. Probar la app **por la pantalla**.

🎯 Lo que sumas hoy

Probar la app **por debajo de la pantalla**: golpear el endpoint HTTP directo. Más rápido, más estable, más barato.

```
// La pirámide del testing

     /\          E2E (UI)    lento, frágil
    /  \         pocos
   /----\        API         ← hoy: rápido
  /      \       muchos            y estable
 /--------\     Unit        los devs
```

🧭 La capa API es el **punto dulce** de QA: cubre lógica de negocio real, sin el costo ni la fragilidad de manejar un navegador.

⭐ QUIZ relámpago · calentamiento

Un **test de API** es, en una frase...

A Un test que abre el navegador y hace clic en botones

B Un test que envía una **petición HTTP** a un endpoint y verifica la **respuesta** (status, body, headers)

C Un test que solo revisa el código fuente sin ejecutarlo

💡 Un test de API habla con el backend en su idioma: `GET`, `POST`, `PUT`, `DELETE` sobre una URL, y comprueba lo que vuelve. Sin pantalla, sin clics — solo petición y respuesta. Hoy aprendemos a escribirlos con Playwright.

request · aserciones · schema · datos · kata

El dolor: abrir un navegador entero para probar un endpoint 🐌

orders.spec.ts — probando la API… por la UI

```ts
test("el pedido se crea", async ({ page }) => {
  await page.goto("/login");              ┐
  await page.fill("#email", EMAIL);       │ 8 segundos de
  await page.fill("#password", PASS);     │ navegador, clics
  await page.click("#login");             │ y esperas…
  await page.goto("/orders/new");         │
  await page.fill("#item", "Coffee");     │
  await page.click("#submit");            ┘
  await expect(page.locator(".toast")).toBeVisible();  // ¿de verdad se guardó?
});
```

🚨 Solo quería saber si `POST /orders` funciona. Pagué **8 segundos**, un login completo y un navegador… y al final confío en un *toast*, no en la respuesta real del servidor.

request · aserciones · schema · datos · kata

La cura: la fixture `request` — HTTP sin navegador 🚀

```
test("POST /orders", async ({ request }) => {
  //                          ▲
  //          otra fixture, como page
  const res = await request.post("/orders", {
    data: { item: "Coffee", qty: 2 },
  });

  await expect(res).toBeOK();    // 2xx
});
```

⚡ Sin `goto`, sin clics, sin esperar render. Una petición directa, una respuesta directa. **Milisegundos**, no segundos.

🎁 `request` es una fixture built-in

Igual que `page`, Playwright te la presta lista. Es un cliente HTTP: `APIRequestContext`.

📨 Habla HTTP directo

`.get()` · `.post()` · `.put()` · `.delete()`. El mismo idioma del backend.

🧬 Cero conceptos nuevos

Pides `{ request }` en vez de `{ page }`. Mismo `await`, mismo `expect`. Solo cambia el canal.

request | aserciones | schema · datos | kata

Anatomía: una petición, una respuesta 📦

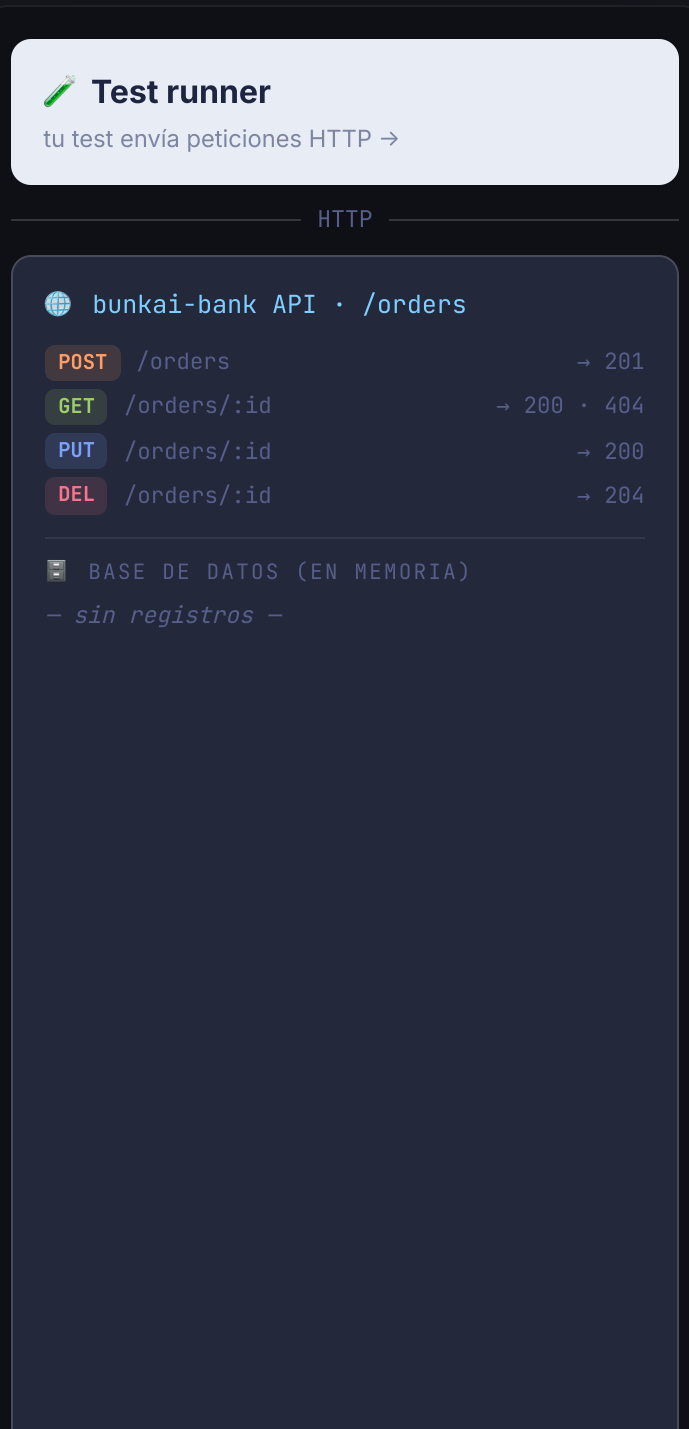
🚩 Los 4 verbos que usarás siempre

```
// leer
await request.get("/orders");
await request.get("/orders/42");

// crear (con cuerpo JSON)
await request.post("/orders", { data });

// actualizar / borrar
await request.put("/orders/42", { data });
await request.delete("/orders/42");
```

📩 Qué te devuelve cada llamada

```
const res = await request.get("/orders/42");

res.status();    // 200, 404, 500…
res.ok();        // true si es 2xx
await res.json();  // el body como objeto JS
res.headers();   // los headers
```

🔑 `status()` y `ok()` son **síncronos**. `json()` es async — el body llega como stream, por eso lleva `await`.



orders-api.spec.ts — clase real, ejecución real

Ejecutable en la versión online

```
class OrdersApi {
  constructor(api) {
    this.api = api;
  }
  async createOrder(data) {
    const res = await this.api.post("/orders", { data });
    expect(res.status()).toBe(201);
    return await res.json();
  }
  async getOrder(id) {
    return await this.api.get("/orders/" + id);
  }
}

test("crea y luego lee un pedido", async ({ api }) => {
  const orders = new OrdersApi(api);
  const created = await orders.createOrder({ item: "Coffee", qty: 2 });

  const res = await orders.getOrder(created.id);
  await expect(res).toBeOK();

  const body = await res.json();
  expect(body.item).toBe("Coffee");
});
```

⭐ QUIZ 1 · La fixture request

Quieres probar `GET /orders` sin tocar la interfaz. ¿Qué fixture pides en tu test?

A `{ page }` — y navegas hasta la lista de pedidos

B `{ request }` — y haces `request.get("/orders")` directo

C `{ browser }` — abres Chrome a mano

💡 `{ request }` es el cliente HTTP: golpea el endpoint sin abrir navegador. Pedir `{ page }` o `{ browser }` arranca un navegador que no necesitas — segundos perdidos en cada test. Para probar la API, habla con la API.

request | **aserciones** | schema · datos | kata

Nuevo dolor: "200 OK" no significa "correcto" ⚠️

```
test("trae el pedido", async ({ request }) ⇒ {
  const res = await request.get("/orders/42");

  await expect(res).toBeOK();    // ✅ 200… ¿y ya?
});

// El server respondió 200, pero el body era:
// { "id": 42, "item": null, "qty": -3 }
// ☠️ datos basura, y el test pasó feliz
```

🕳️ Verificar solo el status es un colador: el endpoint puede responder `200` con un body incompleto, campos `null` o tipos equivocados.

🎯 Tres capas que verificar

1. Status — ¿el código correcto? **2. Body** — ¿los datos correctos? **3. Contrato** — ¿la forma correcta?

🧩 El síntoma

Tests "verdes" que no atrapan nada. Pasan aunque el backend devuelva basura, porque solo miran el status.

💡 La idea

Afirmar sobre el **contenido**: campos exactos, tipos, y la **forma completa** del JSON (schema).

request | **aserciones** | schema · datos | kata

La cura: afirma status, body y headers 🎯

```
test("GET /orders/42 completo", async ({ request }) => {
  const res = await request.get("/orders/42");

  // 1 · STATUS
  expect(res.status()).toBe(200);

  // 2 · BODY (el contenido importa)
  const body = await res.json();
  expect(body.id).toBe(42);
  expect(body.item).toBe("Coffee");
  expect(body.qty).toBeGreaterThan(0);

  // 3 · HEADERS
  expect(res.headers()["content-type"])
    .toContain("application/json");
});
```

🔢 Status exacto

`toBe(201)` para creación, `422` para validación, `404` para no-existe. El código **cuenta una historia**.

📋 Body campo a campo

`toBe` · `toEqual` · `toBeGreaterThan` · `toBeDefined` . Las **mismas** aserciones que ya usas.

📨 Headers cuando importan

content-type, paginación, rate-limit, auth. No siempre, pero cuando el contrato lo exige.

El nivel pro: validar la forma completa con un schema 📐

En vez de afirmar 10 campos a mano, defines la **forma esperada** una vez (con `Zod`) y validas todo el body de golpe. Si falta un campo o cambia un tipo, el schema lo caza.

```
import { z } from "zod";

const OrderSchema = z.object({
  id:   z.number(),
  item: z.string(),
  qty:  z.number().positive(),
});

const body = await res.json();
OrderSchema.parse(body);  // 💥 lanza si la forma no calza
```

🔗 En este repo los schemas se **generan desde la OpenAPI** (`bun run api:sync` → `api/schemas/`). El contrato del backend y tus tests no pueden divergir.

schema-check.js — valida el body en vivo ⚡ Ejecutable en

```
// Mini-validador (idea de Zod, en JS puro)
function checkOrder(body) {
  const errors = [];
  if (typeof body.id !== "number") errors.push("id debe ser nu
  if (typeof body.item !== "string") errors.push("item debe se
  if (typeof body.qty !== "number" || body.qty <= 0)
    errors.push("qty debe ser number positivo");
  return errors;
}

// 👇 Cambia un valor y vuelve a ejecutar
const body = { id: 42, item: "Coffee", qty: 2 };

const errors = checkOrder(body);
if (errors.length === 0) {
  console.log("✓ schema OK — el body respeta el contrato");
} else {
  console.error("✗ schema FALLÓ:");
  errors.forEach(e => console.error("  · " + e));
}
```

🧠 VISUALIZADOR · EL VIAJE DE UNA PETICIÓN

Qué pasa en await request.post(...), paso a paso

```
test("POST /orders", async ({ request }) => {
  const res = await request.post("/orders", {
    data: { item: "Coffee", qty: 2 },
  });
  expect(res.status()).toBe(201);
  const body = await res.json();
  expect(body.id).toBeDefined();
});
```

📍 ¿DÓNDE ESTÁ LA PETICIÓN?

— en reposo —

Presiona "Siguiente paso": vas a seguir la petición desde tu test hasta el servidor y de vuelta.

▶ Siguiente paso ↺ Reiniciar

paso 0 / 6

consola…

⭐ QUIZ 2 · Aserciones

Tu test hace `expect(res).toBeOK()` y pasa. Pero el body llegó `{ item: null, qty: -3 }`. ¿Qué falló en tu test?

A Nada — si el status es 2xx, el endpoint está correcto

B La aserción es débil: faltó verificar el **body** (campos y tipos), no solo el status

C Playwright debió rechazar el body automáticamente

💡 `toBeOK()` solo mira el status. Un 200 con datos basura pasa igual. Una buena aserción de API cubre las tres capas: **status** exacto, **body** (campos + tipos, idealmente con schema) y **headers** cuando el contrato lo pide. Ninguna herramienta adivina qué body es "correcto" — eso lo defines tú.

request · aserciones · **schema · datos** · kata

Nuevo dolor: cada test re-crea su mundo 🔁

```ts
test("actualiza un pedido", async ({ request }) => {
  // 1 · login para sacar token…
  const auth = await request.post("/login", {data:creds});
  const { token } = await auth.json();

  // 2 · crear un pedido para tener qué editar…
  const made = await request.post("/orders", {
    headers: { Authorization: `Bearer ${token}` },
    data: { item: "Coffee", qty: 1 },
  });

  // 3 · …y recién AQUÍ empieza el test real 😩
});
```

🧩 El síntoma

Login + crear precondiciones **antes** de cada test. 15 líneas de ritual por 3 líneas de prueba real.

🔑 El token regado

El `Authorization: Bearer` repetido en cada llamada. Cópialo mal una vez y son 401 misteriosos.

💡 La idea

Una **fixture** que loguee una vez, entregue un cliente ya autenticado y datos frescos — y limpie al final. Lo que viste con `use()`, ahora para API.

request · aserciones · **schema · datos** · kata

La cura: auth una vez + datos únicos por test 🔐

🔑 Cliente autenticado, listo

```ts
export const test = base.extend({
  api: async ({ playwright }, use) => {
    // login UNA vez, reusa el token
    const ctx = await playwright.request.newContext({
      baseURL: BASE,
      extraHTTPHeaders: { Authorization: `Bearer ${TOKEN}` },
    });
    await use(ctx);     // el test lo recibe ya logueado
    await ctx.dispose();
  },
});
```

Todas las llamadas salen ya con el token. Cero `Bearer` regados.

🎲 Datos frescos = tests independientes

```ts
import { faker } from "@faker-js/faker";

// cada test genera SU propio dato único
const order = {
  item: faker.commerce.productName(),
  qty:  faker.number.int({ min: 1, max: 9 }),
};
await api.post("/orders", { data: order });
```

🧰 Datos únicos = ningún test pisa a otro. Es el requisito para poder correr en **paralelo** sin choques.

🎮 SIMULADOR · MISMA COBERTURA, DISTINTA CAPA

6 casos: por UI vs por API — mira el reloj ⏱️

capa de test: 🖥️ UI · E2E | 🔀 Híbrido | ⚡ API puro ⚡ Ejecutable en la versión online

⚡ API puro

POST /orders · 1s | T /orders · 0. | PUT /orders/:id · 1s | E /orders/:id · | 422 validación · 1s | 401 auth · 1s

RELOJ DE PARED	COBERTURA	VELOCIDAD VS UI
5.0s	6 / 6 casos	5.80×

⭐ QUIZ 3 · Datos y velocidad

¿Por qué cada test de API debe generar sus datos con `faker` en vez de usar un id fijo como `"order-1"`?

A Porque faker hace los tests más cortos de escribir

B Para que sean **independientes**: con datos únicos ningún test pisa a otro y pueden correr en paralelo

C Porque Playwright exige datos aleatorios obligatoriamente

💡 Datos fijos compartidos = colisiones: dos tests crean `"order-1"` y uno rompe al otro. Datos únicos por test = aislamiento total, y el aislamiento es lo que permite paralelizar sin flakiness. La velocidad de API solo se aprovecha si los tests no se estorban entre sí.

request · aserciones · schema · datos · **kata**

KATA: todo lo de hoy, ordenado en capas 🥋

KATA (**Component Action Test Architecture**) no inventa nada: **organiza** el cliente, las aserciones y los datos en 4 capas, donde cada una `extends` la anterior. El lado API es idéntico al UI — solo cambia la base.

🎁 Cliente → Componentes

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

🎯 Aserciones → dentro de la ATC

Status y schema fijos van **dentro**; las de negocio, en el test.

🔧 Auth + datos → Fixtures (L4)

`{ api }` entrega el cliente autenticado y listo.

```
// La cadena de herencia REAL del repo (API)

class TestContext { ... }        // L1 · config, faker
class ApiBase extends TestContext // L2 · helpers HTTP
class OrdersApi extends ApiBase // L3 ← tu cliente

// y la fixture lo inyecta:
class ApiFixture {
  orders = new OrdersApi(options);
}
```

🧬 Misma arquitectura que el lado UI: solo cambia `ApiBase` por `UiBase`. Aprendes una vez, aplicas a ambos.

request · aserciones · schema · datos · **kata**

Las 4 capas de KATA, lado API 🏛️

L4 · Fixtures (DI) — ApiFixture · TestFixture — inyectan el cliente autenticado

↑ usa

L3.5 · Steps — cadenas de ATCs reutilizables (precondiciones)

↑ usa

⭐ L3 · Componentes de dominio — **OrdersApi · UsersApi — aquí viven las ATCs**

↑ extends

L2 · ApiBase — helpers HTTP: get/post/put/del + parseo + auth

↑ extends

L1 · TestContext — config · logger · faker · entorno (compartido con UI)

📐 Regla única: una capa superior puede usar una inferior, **nunca** al revés. Tu cliente de API (L3, estrella) es el corazón — escribes ATCs ahí y todo lo demás ya viene hecho.

request · aserciones · schema · datos · **kata**

La pieza nueva: la ATC de API 🎯

```
class OrdersApi extends ApiBase {

  @atc("PROJ-201")
  async createOrder(data) {
    const res = await this.post("/orders", { data });

    // aserciones FIJAS dentro:
    expect(res.status()).toBe(201);
    const body = await res.json();
    OrderSchema.parse(body);   // contrato
    return body;
  }
}
```

🎯 ATC = un caso completo

Acceptance **T**est **C**ase: no una llamada suelta, sino un **mini-flujo** — request → status fijo → schema → devuelve datos.

`@atc("PROJ-201")`

El decorador ata la ATC a su **ticket** de Jira 1:1. Trazabilidad y tracing automático.

⚖️ Fijas dentro, negocio fuera

Status `201` y schema (siempre ciertos) van **dentro**. Las aserciones de negocio van en el **test**.

🧠 TRAZADOR · SIGUE UNA LLAMADA POR LAS 4 CAPAS

Un test llama api.orders.createOrder() — ¿por dónde viaja?

```
test("PROJ-201: crea pedido", async ({ api }) => {
  await api.orders.createOrder(data);
});

// L4 ApiFixture.orders = new OrdersApi(opts)
// L3 class OrdersApi extends ApiBase
// L2 class ApiBase extends TestContext
// L1 class TestContext { config, faker, request }
```

▶ Siguiente paso ↺ Reiniciar paso 0 / 5

🏛 CAPA ACTIVA

— en reposo —

Presiona "Siguiente paso": vas a seguir la llamada bajando por las 4 capas de KATA.

request · aserciones · schema · datos · **kata**

En la práctica: pides la fixture según lo que pruebas 🎚️

TIPO DE TEST	FIXTURE	¿NAVEGADOR?	CUÁNDO
API pura (integration)	`{ api }`	No	Testing de API. Default en `tests/integration/`
Solo UI	`{ ui }`	Sí	Flujo de UI sin setup por API
Híbrido	`{ test }`	Sí	Setup por API + acción/verificación UI
Precondiciones repetidas	`{ steps }`	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.createOrder(data);
  await test.ui.orders.verifyVisible(o.id);
});
```

⭐ QUIZ 4 · KATA

En una ATC `createOrder()`, ¿qué aserción va **dentro** de la ATC y cuál en el **test**?

A **Dentro:** status `201` + schema (siempre ciertos). **En el test:** "el total facturado es 14€" (regla de negocio)

B Todas dentro de la ATC — así el test queda más corto

C Todas en el test — la ATC solo debe hacer la llamada HTTP

💡 Aserciones **fijas e inevitables** (un 201 al crear, el schema del contrato) son parte de "la acción terminó bien" → van dentro. Las de **negocio**, que cambian por caso (el total, el descuento, quién puede verlo) → van en el test, visibles. Misma regla que en el lado UI: efectos dentro, veredicto fuera.

⭐ QUIZ 5 · El integrador

Conecta cada dolor con su cura:

A navegador para un endpoint → schema · status débil → faker · setup repetido → request

B navegador para un endpoint → **request** · "200" no basta → **aserciones + schema** · setup repetido → **fixtures + faker**

C todos los dolores → un solo truco mágico

💡 Cada herramienta cura un dolor: **request** habla HTTP sin navegador, **aserciones + schema** verifican contenido y contrato, **fixtures + faker** ordenan auth y datos. Y **KATA** apila todo en capas para que escale. Cuatro herramientas, cuatro problemas.

DEL CONCEPTO AL REPOSITORIO

Dónde vive cada pieza en este repo 🗂️

```
tests/
  components/
    TestContext.ts        // L1
    api/ApiBase.ts        // L2 · helpers HTTP
    api/OrdersApi.ts      // L3 · cliente + ATCs
    steps/*.ts            // L3.5 · Steps
    ApiFixture.ts         // L4 · Fixtures
    TestFixture.ts        // L4 · híbrido
  integration/**          // tests API puros
  e2e/**                  // tests UI (+API)
api/schemas/              // Zod desde OpenAPI
playwright.config.ts      // baseURL, workers
```

📍 Tu zona de trabajo

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

🛠️ Ya hecho por ti

TestContext, ApiBase y las Fixtures vienen en el boilerplate. Heredas helpers; no los reescribes. Los schemas se generan: `bun run api:sync`.

📖 Para profundizar

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

Cuatro herramientas, un mapa 🗺️

HERRAMIENTA	DOLOR QUE CURA	PIEZA CLAVE	EN KATA
🚀 request fixture	navegador para un endpoint	`request.post(...)`	ApiBase (L2)
🎯 Aserciones + schema	"200" no significa correcto	`status + body + Zod`	dentro de la ATC (L3)
🔧 Fixtures + faker	setup/auth/datos repetidos	`{ api }` + datos únicos	Fixtures (L4)
🥋 KATA	todo junto, a escala	4 capas + ATC	el marco completo

🏆 Cruzaste de **"probar por la pantalla"** a **probar el backend directo**: más rápido, más estable, más profundo. Lo que hace un SDET de API.

MISIÓN CUMPLIDA

Tu puntaje final

— / 6

Re-juega en casa

Este archivo corre offline: vuelve al runner de API, al validador de schema y al simulador de velocidad. Toca todo.

Reto puente

Abre `tests/components/api/` del repo y ubica: capa, ATC, decorador `@atc`, aserciones fijas, schema.

Próximo paso

Escribe tu primera ATC de API real con `/test-automation` — Plan → Code → Review sobre KATA.

Hoy: `request` → aserciones + schema → fixtures + faker → **KATA**. De la pantalla al backend.

QA ENGINEERING · API AUTOMATION

De la pantalla al backend 

request · aserciones · schema · KATA.
Probar la API: rápido, estable y al hueso.

¿Preguntas? /test-automation references/kata-architecture.md