CLASE ESPECIAL · QA ENGINEERING

Coding Base

Fundamentos de programación para QA Testers.
De tester manual a **Quality Engineer**, un concepto a la vez.

JavaScript · TypeScript · Playwright · 8 quizzes ⭐ · playgrounds en vivo 🎮

SPOILER

Ya sabes programar. Solo que en papel.

TEST CASE MANUAL	SCRIPT AUTOMATIZADO
Pasos numerados (1, 2, 3...)	Líneas de código (se ejecutan en orden)
Datos de prueba ("usuario: ana@qa.com")	Variables (`const email = "ana@qa.com"`)
Resultado esperado	`expect(...)` — la aserción
Precondiciones	Setup / hooks
Lo ejecuta un humano 🧍	Lo ejecuta la máquina 🤖 (miles de veces, sin café)

💡 **Un script es un test case que la máquina ejecuta por ti.** Hoy solo aprendemos el idioma para escribirlo.

EL PLAN

Una sola idea: zoom-out infinito 🔍

Empezamos en un dato diminuto y alejamos la cámara slide a slide hasta ver un repositorio completo.

⚛️

Acto 1 · Átomos

Valores, tipos, variables, operadores.

🧠

Acto 2 · Lógica

Funciones, decisiones, loops, datos.

🛡️

Acto 3 · TypeScript + async

Tipos que te protegen. Esperas elegantes.

🎭

Acto 4 · Tu primer test

Anatomía Playwright + playground en vivo.

🛰️

Acto 5 · Vista satelital

El repo completo. Ejecutar y leer resultados.

⭐

Reglas del juego

8 quizzes. Tu puntaje vive arriba a la derecha. Los playgrounds se tocan.

valor variable función lógica archivo repo

ZOOM NIVEL 1

Todo empieza con un dato

"Ana"

Esto es un **valor**: la pieza más pequeña de información que un programa puede manejar. Las comillas dicen "esto es texto". Sin contexto, sin nombre, sin lógica. Un átomo.

`valor` `variable` `función` `lógica` `archivo` `repo`

Los valores tienen tipos

📝

string

```
"Ana"
"ana@qa.com"
"Login exitoso"
```

Texto. Siempre entre comillas.

🔢

number

```
25
3.14
404
```

Números. Sin comillas. `404` ≠ `"404"`

🚦

boolean

```
true
false
```

Verdadero o falso. El tipo favorito de QA: **¿pasó o falló?**

🧪 Analogía QA: los campos de un formulario. Nombre = string, edad = number, "acepto términos" = boolean.

valor | **variable** | función | lógica | archivo | repo

Variables: cajas con etiqueta 📦

```
const nombre = "Ana";
const edad = 25;
let intentos = 0;
```

```
const/let  → "voy a crear una caja"
nombre     → la etiqueta de la caja
=          → "guarda esto adentro"
"Ana"      → el valor guardado
```

🔒 const

Caja sellada. El valor **no se puede reemplazar**. Tu opción por defecto.

🔓 let

Caja reutilizable. El valor **puede cambiar** (contadores, reintentos).

🧪 En testing

`const email = "ana@qa.com"`

Tus datos de prueba con nombre propio.

🎮 PLAYGROUND EN VIVO

Tócalo. Rómpelo. Arréglalo.

variables.js — editable

⚡ Ejecutable en la versión online

```
// Cambia los valores y vuelve a ejecutar
const tester = "Ana";
let bugsFound = 3;

console.log("Tester: " + tester);

bugsFound = bugsFound + 1;
console.log("Bugs encontrados: " + bugsFound);

// Reto 1: crea una variable "rol" y muéstrala
// Reto 2: intenta reasignar tester = "Luis"  (¿qué pasa?)
```

⭐ QUIZ 1 · Variables y tipos

¿Cuál de estas líneas lanza un ERROR?

A `let retries = 1; retries = 2;`

B `const user = "Ana"; user = "Luis";`

C `const passed = true;`

💡 **const** crea una caja sellada: reasignarla lanza `TypeError: Assignment to constant variable`. **let** sí permite cambiar el valor.

valor · **variable** · función · lógica · archivo · repo

Operadores: hacer y comparar

➕ Matemáticos

```
5 + 3        // 8
10 - 4       // 6
"QA" + "!"   // "QA!"
```

Con strings, `+` une texto.

⚖️ Comparación

```
5 === 5      // true
5 !== 3      // true
10 > 7       // true
```

Siempre devuelven **boolean**.

🔗 Lógicos

```
a && b   // Y: ambas
a || b   // O: alguna
!a       // NO: invierte
```

Combinan condiciones.

⚠️ **La trampa #1:** `=` asigna (guarda en la caja) · `===` compara (¿son iguales?). Confundirlos es el bug clásico de principiante.

⭐ QUIZ 2 · Operadores

¿Qué imprime `console.log(5 === "5")`?

A `true` — cinco es cinco

B `false` — número y texto no son lo mismo

C Error — no se pueden comparar

💡 `===` compara valor **y tipo**. `5` es number, `"5"` es string → `false`. En testing esto importa: un API que devuelve `"404"` (texto) no es lo mismo que `404` (número).

valor | variable | **función** | lógica | archivo | repo

Funciones: recetas con nombre 🍳

```
function login(email, password) {
  const mensaje = "Bienvenida " + email;
  return mensaje;
}

const resultado = login("ana@qa.com", "secreto123");
```

function

"Voy a definir una receta."

login(email, password)

Nombre + **parámetros**: los ingredientes que recibirá.

return

Lo que la receta **entrega** al terminar.

login("ana@qa.com", ...)

Llamarla = ejecutar la receta con ingredientes reales.

¿Por qué te importan? Reutilización

😫 Tester manual

TC-101: abrir login, escribir email, escribir password, click...
TC-102: abrir login, escribir email, escribir password, click...
TC-103: abrir login, escribir email... **otra vez. y otra. y otra.**

😎 Con una función

```
login("ana@qa.com", "secreto123");
login("luis@qa.com", "otraClave9");
login("mia@qa.com", "claveMia55");
```

Escribes los pasos **una vez**, los ejecutas con mil datos distintos.

💡 Una función es un **test step reutilizable**. Cuando el botón de login cambie, corriges UN lugar — no 40 test cases.

🎮 PLAYGROUND EN VIVO

Tu primera función

functions.js — editable

⚡ Ejecutable en la versión online

```
function greet(name) {
  return "Hola, " + name + " 👋";
}

console.log(greet("Ana"));
console.log(greet("Luis"));

// Reto 1: llama a greet con TU nombre
// Reto 2: crea una función farewell(name) que
//         devuelva "Chau, <name>" y úsala
```

★ QUIZ 3 · Funciones

¿Qué imprime este código?

```
function sum(a, b) {
  return a + b;
}
console.log(sum(2, 3));
```

A `5` — suma los dos numbers

B `"23"` — los pega como texto

C `undefined` — falta algo

💡 2 y 3 son **numbers** (sin comillas) → + suma: 5. Si fueran `"2"` y `"3"` (strings), + los pegaría: `"23"`. El tipo decide el comportamiento.

valor · variable · función · **lógica** · archivo · repo

if / else: el código decide 🔀

```
if (statusCode === 200) {
  console.log("✅ Test PASSED");
} else {
  console.log("❌ Test FAILED");
}
```

🤔 La pregunta

`statusCode === 200` se evalúa a **true** o **false** (¡los booleans del Acto 1!).

✅ true → primer bloque

❌ false → bloque del `else`. Nunca ambos.

🧪 ¿Te suena?

Es tu **resultado esperado**: "SI el sistema muestra X, el test pasa; SI NO, falla". Llevas años escribiendo if/else en español.

valor · variable · función · **lógica** · archivo · repo

Loops: repetir sin sufrir 🔁

El clásico: `for`

```
for (let i = 0; i < 3; i++) {
  console.log("Intento #" + i);
}
// Intento #0, #1, #2
```

Tres piezas: **inicio** (`i = 0`) · **condición** (`i < 3` : ¿sigo?) · **paso** (`i++` : suma 1).

El elegante: `for ... of`

```
const users = ["Ana", "Luis", "Mia"];

for (const user of users) {
  login(user);
}
```

"Por cada user de la lista, haz login." Esto es **data-driven testing**: mismos pasos, N datos.

🧠 VISUALIZADOR DE MEMORIA

El loop, paso a paso

```
const users = ["Ana", "Luis", "Mia"];
for (let i = 0; i < users.length; i++) {
  console.log("Login de " + users[i]);
}
```

▶ Siguiente paso ↺ Reiniciar paso 0 / 9

📦 MEMORIA

— vacía —

Presiona "Siguiente paso" y mira cómo la máquina ejecuta línea por línea.

consola…

⭐ QUIZ 4 · Condicionales y loops

¿Qué imprime `for (let i = 0; i < 3; i++) console.log(i)`?

A `0 1 2`

B `1 2 3`

C `0 1 2 3`

💡 `i` empieza en **0** (los programadores cuentan desde cero) y el loop corre mientras `i < 3` — cuando i llega a 3, la condición es false y se detiene **sin** imprimir el 3.

valor | variable | función | **lógica** | archivo | repo

Arrays: listas ordenadas 📋

```
const browsers = ["chrome", "firefox", "safari"];

browsers[0]      // "chrome"  ← ¡desde cero!
browsers[2]      // "safari"
browsers.length  // 3

browsers.push("edge");  // agrega al final
```

🔢 Índice

Cada posición tiene número, **empezando en 0**.

El primero es `[0]`, no `[1]`.

📏 .length

Cuántos elementos hay. Lo usaste en el loop: `i < users.length`.

🧪 En testing

Una **columna de tu tabla de test data**: lista de emails, de browsers, de roles a probar.

valor | variable | función | **lógica** | archivo | repo

Objetos: fichas con campos 📦

```
const user = {
  email: "ana@qa.com",
  password: "secreto123",
  role: "admin",
  active: true,
};

user.email    // "ana@qa.com"
user.role     // "admin"
```

🏷️ Propiedades

Pares **nombre: valor**. Como un formulario relleno: cada campo tiene etiqueta y contenido.

📍 El punto

`user.email` = "del objeto user, dame email". Lo verás MIL veces:

`page.click` es exactamente esto.

🧪 Combo final

Array de objetos = **tu tabla de test data completa**: cada objeto una fila, cada propiedad una columna.

🎮 PLAYGROUND EN VIVO

Todo junto: datos + loop

test-data.js — editable ⚡ Ejecutable en la versión online

```
const users = [
  { name: "Ana", role: "admin" },
  { name: "Luis", role: "viewer" },
];

for (const u of users) {
  if (u.role === "admin") {
    console.log(u.name + " puede borrar proyectos 🔑");
  } else {
    console.log(u.name + " solo puede mirar 👀");
  }
}

// Reto: agrega un tercer usuario con role "editor"
//       y dale su propio mensaje con else if
```

★ QUIZ 5 · Arrays y objetos

¿Qué imprime este código?

```
const fruits = ["uva", "kiwi", "mango"];
console.log(fruits[1]);
```

A `"uva"` — el primero

B `"kiwi"` — índice 1 = segunda posición

C `"mango"` — el último

💡 Los arrays cuentan desde **0**: `fruits[0]` es "uva", `fruits[1]` es "kiwi". Error clásico de todo principiante (y de varios seniors a las 6pm).

NIVEL SIGUIENTE

TypeScript = JavaScript + red de seguridad 🛡️

login.ts — Visual Studio Code

```ts
function login(email: string, password: string) {
  // ...
}

login("ana@qa.com", 12345);
```

⛔ Argument of type 'number' is not assignable to parameter of type 'string'.

login.ts(5, 22) — ts(2345)

🏷️ Tipos explícitos

`email: string` declara qué tipo acepta cada parámetro. El contrato queda escrito.

⚡ Error ANTES de ejecutar

El subrayado rojo aparece **mientras escribes** — no en producción, no en el test run de las 3am.

🧪 ¿Te suena el concepto?

Encontrar el defecto lo más temprano posible... **TypeScript es shift-left para tu propio código.** 😉

Tipos que usarás desde el día 1

```ts
const email: string = "ana@qa.com";
const retries: number = 3;
const passed: boolean = true;

interface User {
  email: string;
  role: "admin" | "viewer";
}
```

📐 Anotación

`: tipo` después del nombre. La mayoría de veces TypeScript lo **deduce solo** y ni lo escribes.

📋 interface

El "plano" de un objeto: qué campos y de qué tipo. Tu test data con contrato.

🚧 Union: `"admin" | "viewer"`

Solo estos valores exactos. Escribes `"admni"` → error al instante. Adiós typos silenciosos.

async / await: saber esperar ⏳

El problema

La web es **lenta**: la página carga, el servidor responde, la animación termina... Tu código corre en **milisegundos**; la web responde en **segundos**.

```
// 🤖💥 robot impaciente (SIN await):
page.goto("https://app.com/login");
page.click("#login-btn");
// click sobre una página que AÚN NO CARGÓ ❌
```

La solución

```
async function runTest() {
  await page.goto("https://app.com/login");
  await page.click("#login-btn");
}
```

`await` = "espera a que esto **termine** antes de seguir".

`async` = "esta función contiene esperas".

📜 **Regla de oro Playwright:** casi toda acción lleva `await`. Olvidarlo = tests intermitentes (flaky).

★ QUIZ 6 · TypeScript + async

¿Por qué escribimos `await page.click("#save-btn")`?

A Decoración — el test corre igual sin await

B Hace que el click sea más rápido

C El click tarda — el código debe **esperar a que termine** antes de seguir

💡 Sin `await`, el código sigue corriendo sin esperar el click → el siguiente paso actúa sobre una página a medio cambiar → test **flaky** (a veces pasa, a veces no). El await olvidado es la causa #1 de tests intermitentes.

Anatomía de un test real — ya conoces cada pieza

login.spec.ts

```ts
import { test, expect } from "@playwright/test";

test("login exitoso", async ({ page }) => {
  await page.goto("https://app.com/login");
  await page.fill("#email", "ana@qa.com");
  await page.fill("#password", "secreto123");
  await page.click("#login-btn");

  await expect(page.locator("#welcome")).toBeVisible();
});
```

📦 import

"Tráeme las herramientas `test` y `expect` de la caja Playwright."

🧪 test("...", async ...)

¡Una **función** que recibe el nombre del caso... y otra función con los pasos! Sí: las funciones viajan como valores.

🤖 page.fill / page.click

Funciones con parámetros (Acto 2) + el punto de objetos (Acto 2) + await (Acto 3).

✅ expect(...).toBeVisible()

Tu **resultado esperado** hecho código. Si no se cumple → test FAILED.

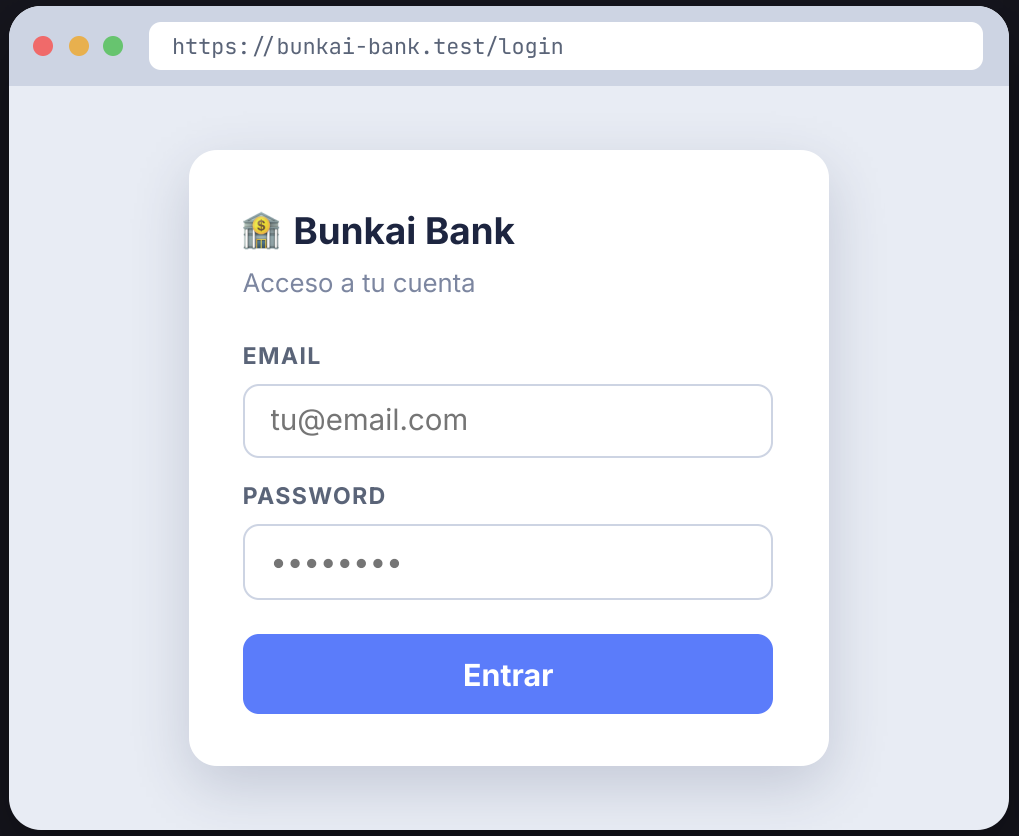
Locators: cómo el robot "ve" 👁️

TÚ VES...	PLAYWRIGHT NECESITA...
"el campo de email"	`page.fill("#email", …)` — por su **id** (#)
"el botón azul de Login"	`page.getByRole("button", { name: "Login" })` — por su rol visible
"el mensaje de bienvenida"	`page.locator("#welcome")` — apunta al elemento, listo para verificar

🕵️ Cada elemento de una página tiene "dirección": su **selector**. Encontrarla es destreza detective de QA — y la vas a practicar AHORA mismo.

🎮 PLAYGROUND · MODO DETECTIVE

Inspecciona la app — pasa el mouse 🔍



1️⃣ Activa el modo inspección

Y pasa el mouse sobre la mini-app.

2️⃣ Misión

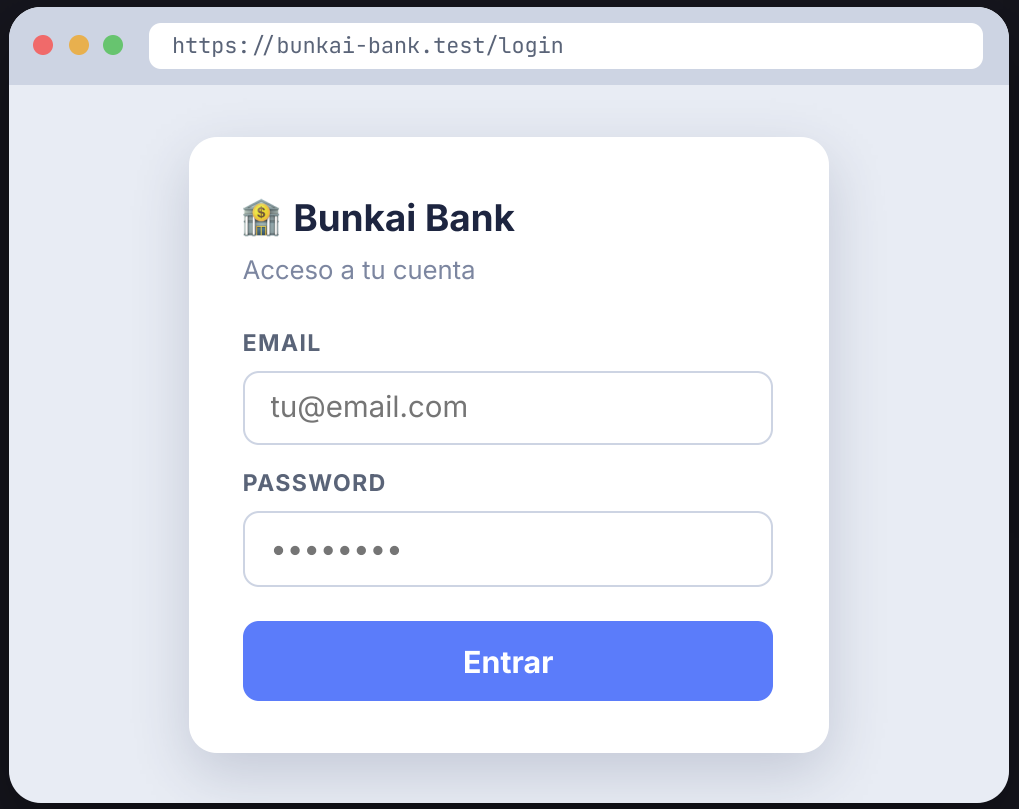
Encuentra el selector del campo email, del password y del botón.
Apúntalos: los usarás en la próxima slide.

💡 En la vida real

Esto mismo se hace con **DevTools** (click derecho → Inspeccionar) o con el inspector de Playwright.

🎮 PLAYGROUND · EL GRAN MOMENTO

Tu primer test, ejecutándose en vivo 🤖



mi-primer-test.spec.ts — ⚡ Ejecutable en la versión online

```ts
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();
});
```

▶ *Ejecuta el test y mira el cursor fantasma…*

⭐ QUIZ 7 · Anatomía Playwright

¿Qué hace `await page.fill("#email", "ana@qa.com")`?

A Hace click en el campo email

B Verifica que el campo contiene ese texto

C Escribe `"ana@qa.com"` dentro del elemento con id `email`

💡 `fill` = rellenar: primer argumento el **selector** (dónde), segundo el **texto** (qué). Verificar contenido sería `expect(...).toHaveValue(...)` — las acciones actúan, los **expect** verifican.

valor · variable · función · lógica · archivo · **repo**

Zoom final: del átomo al repositorio 🛰️

```
mi-proyecto/
├─ package.json          ← la "cédula": nombre,
dependencias, comandos
├─ playwright.config.ts  ← configuración:
browsers, URL base, timeouts
├─ node_modules/         ← las herramientas instaladas
(no se toca)
└─ tests/
   ├─ login.spec.ts      ← ⭐ tu test vive aquí
   └─ signup.spec.ts     ← cada feature, su archivo
```

🔭 El viaje completo

"Ana" — un valor
↳ const email — con nombre
↳ login(email, pass) — dentro de una función
↳ if / for — con lógica
↳ login.spec.ts — en un archivo
↳ tests/ — en una carpeta
↳ mi-proyecto/ — en un repositorio 🎉

Un repo es solo **carpetas y archivos organizados**. Nada de magia: cada nivel lo construiste hoy.

Ejecutar y leer el veredicto 🏁

✅ Cuando todo pasa

```
$ npx playwright test

Running 1 test using 1 worker

  ✓ login.spec.ts:3 › login exitoso (2.1s)

  1 passed (2.5s)
```

❌ Cuando algo falla — lee con calma

```
  ✗ login.spec.ts:3 › login exitoso

  Error: expect(locator).toBeVisible()
  Locator: locator("#welcome")
  Expected: visible
  Received: hidden
    at login.spec.ts:8:48
```

El error te dice **qué** esperaba, **qué** encontró y **en qué línea**. Leer errores con calma = tu superpoder QA.

★ QUIZ 8 · El integrador final

A este test le falta UNA palabra. ¿Cuál?

```
test("guardar cambios", async ({ page }) => {
  page.click("#save-btn");   // ⚠️ algo falta aquí…
  await expect(page.locator("#saved-msg")).toBeVisible();
});
```

A `return` antes de `page.click`

B `await` antes de `page.click` — hay que esperar el click

C `const` antes de `page.click`

💡 Sin `await`, el test NO espera el click y corre directo al `expect` → el mensaje aún no existe → falla intermitente (flaky). Acabas de diagnosticar el bug más común de la automatización. **Bienvenida/o al oficio.** 🎓

MISIÓN CUMPLIDA

Tu puntaje final

— / 8

1️⃣

Instala el kit

VS Code + Node.js. 10 minutos, gratis.

2️⃣

Crea tu proyecto

```
npm init playwright@latest
```

y el árbol de la slide 32 aparece solo.

3️⃣

Continúa el roadmap

Siguiente etapa — **Dojo Lab 1**: tus primeros specs reales, en la Dojoteca. 🚀

Ir a los Decks del DOJO →

Hoy: valores → variables → funciones → lógica → TypeScript → async → tu primer test. **Ya hablas el idioma.**