> how-it-works · agentic-qa · sprint-testing

La Trifuerza — Probar Features por Capas

UI · API · DB. Pruebes lo que pruebes en una feature o historia de usuario, piensas SIEMPRE en las tres capas: una pantalla verde puede esconder una API rota y un trigger que falla en silencio. Smoke primero, luego las tres capas, con sondas de seguridad e integridad.

Smoke → UI → API → DB

Tu lente al probar cualquier feature en el sprint

// la verdad que duele

201 ≠ DB ✓

Un **201 Created** no prueba que la base de datos quedó correcta — los triggers pueden no ejecutarse en silencio. La capa que **responde** bien no es la capa que **persiste** bien. Por eso se prueban las tres.

// qué cubriremos

Probar Toda Feature, por Capas

3 quizzes interactivos a lo largo del recorrido — haz clic en las respuestas.

// el modelo

Tres Capas, Tres Verdades

UI

Lo que el usuario ve. ¿Renderiza, fluye, da feedback? Puede mostrar datos cacheados o mock.

API

El contrato. ¿Status correcto, schema válido, auth y RLS? El 2xx no garantiza la DB.

DB

La verdad persistida. ¿La fila existe, el trigger corrió, los constraints aguantan?

TIPO DE FEATURE	ORDEN DE CAPAS
UI-focused	`UI → API → DB`
API-first	`API → DB → UI`
Data-focused	`DB → API → UI`
Full-stack	`las tres`

// tres ejes, no uno

Capas · Niveles · Tipos

La Trifuerza es UN eje: **las capas — dónde sondeas** (UI · API · DB). Es **ortogonal** a otros dos ejes que el tester decide en cada prueba. No compiten: se combinan.

Capas — ¿dónde sondeas?

La Trifuerza. UI · API · DB. Cada feature se atraviesa por sus tres capas, siempre.

Niveles — ¿qué alcance?

component → integration → end-to-end (e2e). Cuánto del sistema entra en la prueba.

Tipos — ¿con qué intención?

funcional · regresión · smoke · seguridad · performance · usabilidad…

Cómo se cruzan

Una misma historia de usuario se prueba **eligiendo un nivel y un tipo** — pero **SIEMPRE atravesando las tres capas**. Ej.: un e2e funcional del checkout y una prueba de seguridad de la API distinta en nivel y tipo, idénticas en que ambas piensan UI/API/DB.

EL GATE

Smoke First
Valida el Ambiente, no la Feature

5-10 minutos de Go/No-Go. Nunca entres a probar a fondo hasta que el smoke pase. Un ambiente roto produce reportes de bug falsos.

// en orden, ≤10 min

El Checklist de Smoke

1. **Acceso básico**
   La app carga, sin 500, assets renderizan, consola sin errores rojos (warnings amarillos OK).

2. **Autenticación**
   Login, la sesión persiste al refrescar, logout funciona (saltar si el ticket es pre-auth).

3. **Happy path**
   El flujo primario del ticket de punta a punta, 3-5 pasos, marca cada uno al funcionar.

4. **Integración backend**
   Network tab durante el happy path: todas las llamadas `/api/*` en 2xx, los datos persisten tras F5.

> **SIEMPRE revisa el Network tab — aunque la UI se vea bien**
>
> La UI puede renderizar datos cacheados o mock. La red no miente: si la UI muestra algo pero no hubo llamada 2xx, no confíes en la pantalla.

// el veredicto y sus reglas

Go / No-Go

✓ PASS → continúa

- Registra `Smoke: PASSED` en memory
- Procede a UI / API / DB
- Captura evidencia del home

vs

✗ FAIL → STOP

- Es un bloqueante de ambiente
- Triagea primero (NO auto-Critical)
- Surfacea al usuario, NO pruebes a fondo

Las restricciones del smoke

Durante el smoke: **no** corras edge cases ni negativos · **no** abras bugs menores (solo bloqueantes) · **no** excedas 10 minutos. El smoke detecta bloqueantes, no documenta calidad.

// dos preguntas distintas

Smoke ≠ Testing Profundo de la Feature

Smoke pregunta...

"¿El **ambiente** está vivo?" — ¿carga, autentica, persiste el happy path? Si no, todo lo demás es ruido.

La Trifuerza pregunta...

"¿La **feature** funciona bien?" — ¿cumple los ACs, aguanta edge cases, es segura e íntegra en sus tres capas?

El orden importa

Un ambiente roto produce **reportes de bug falsos**. Si smoke falla y pruebas igual, vas a reportar "bugs" que en realidad son el ambiente caído. Smoke valida el ambiente; la Trifuerza valida la feature. Primero uno, luego la otra.

CAPA 01 · UI

La Capa UI
Lo que el Usuario Ve

Valida los ACs, descubre edge cases, captura evidencia — y desconfía de la pantalla: puede esconder un fallo de la capa de abajo.

// el loop por cada AC

Navega → Observa → Captura

1

Navega + snapshot

Abre la URL de inicio, captura la estructura del DOM / árbol de accesibilidad.

2

Ejecuta + observa

Acciones del happy path paso a paso. ¿Resultado esperado? ¿estado UI raro? ¿consola roja? ¿red 4xx/5xx?

3

Captura + propaga estado

Screenshot en el estado clave (`{KEY}-{n}-ac{N}-desc.png`) y marca el outline PASSED/FAILED en memory.

Reglas de DevTools

Consola: **rojo = log** (amarillo OK). Red: filas **rojas (4xx/5xx) siempre se anotan** aunque la UI se vea bien — la UI puede enmascarar un fallo de API. >5s de carga = hallazgo de Performance. Desactiva el cache.

// el checklist por input / interacción

Edge Cases de la Capa UI

CATEGORÍA	QUÉ PROBAR
Boundary (BVA)	vacío · `min-1` · `min` · `min+1` · `max-1` · `max` · `max+1` · 0 / -1 / MAX_INT · especiales `<script>`, `'; DROP TABLE`
UI / sesión	refrescar a mitad de flujo · botón atrás · pestañas duplicadas · timeout / idle
Estado / transición	cada transición válida, luego una que el estado debe **rechazar**
Validación de datos	email inválido · password débil · doble submit · edición concurrente
Visual	breakpoints responsive · loading states · layouts rotos · elementos solapados

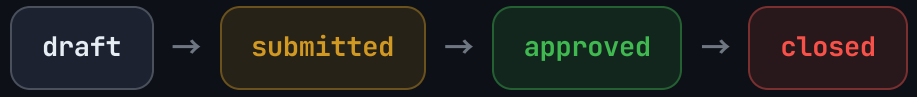
Si al probar encuentras un caso nuevo...

...que el ATP de Stage 1 no derivó (una partición, un boundary, una transición), **añádelo de vuelta** al set de outlines. La cobertura es el piso (ACs) **más** esta capa de riesgo, no el testing improvisado en lugar de la planificación.

// donde se esconden los defectos

Transiciones Inválidas — el Punto Ciego

Para cualquier entidad con estado (draft→submitted→approved, cart→paid→shipped): dispara cada transición **válida**, luego dispara una que el estado actual **debería rechazar**.



la sonda: `approve(closed) → ¿rechazado?` un 200 silencioso sin cambio de estado = bug

> **El happy path casi siempre funciona**
>
> Lo que rompe es cuando alguien intenta hacer lo que el negocio no permite — aprobar algo ya cerrado, cancelar algo ya enviado. **Las transiciones inválidas son donde viven los bugs de máquina de estados.**

// cuándo bajar una capa

Cuándo Escalar a API / DB

UI renderiza pero el dato se ve mal

→ baja a §DB para confirmar qué se persistió de verdad.

Un botón dispara un request non-2xx

→ baja a §API para reproducir la llamada directamente.

Sospecha de RLS — ves filas que no deberías

→ baja a §API y §DB ambas.

La UI es el síntoma, no el diagnóstico

Cuando algo se ve mal arriba, la causa casi siempre está abajo. Escalar de capa es cómo pasas de "esto se ve raro" a "esto está roto aquí, por esto".

CAPA 02 · API

La Capa API
El Contrato

Confirma contratos, auth, manejo de errores y RLS. Cada status code no es un número — es una prueba de que algo funciona (o no).

// descubrir + autenticar

Preparar el Testing de la Capa API

auth una vez, reutiliza

```
# credenciales SIEMPRE desde .env — nunca hardcodear
POST /auth/v1/token
  body: { email: {{env.STAGING_USER_EMAIL}}, password: {{env.STAGING_USER_PASSWORD}} }
  store: access_token, refresh_token, user.id

# registra los endpoints del ticket en una tabla en memory:
# | Method | Endpoint | Purpose | AC |
```

Gotcha de propagación de token

Algunos backends scopean el token por workspace/tenant. Si un request da **200 para un usuario y 403 para otro con el mismo rol**, la propagación es la sospechosa — anótalo como **descubrimiento**, no como bug, hasta confirmarlo con el equipo.

// cada status PRUEBA algo

La Matriz de Status Codes

ESCENARIO	ESPERADO	QUÉ PRUEBA
Happy path	`200 / 201`	Contrato + data feliz
Campo requerido faltante	`400`	Validación viva
Input sobre-largo	`400 / 413`	Guardia de tamaño
Sin header de auth	`401`	Auth requerida
Token expirado	`401`	La rotación de token funciona
Recurso de otro tenant	`403 / vacío`	**RLS / aislamiento de tenant**
Recurso inexistente	200+vacío (PostgREST) / 404	Semántica de not-found
Único duplicado	`409`	Manejo de conflicto

// la sonda de seguridad que casi nadie corre

Sonda de RLS / Autorización — A/B

Crítica para apps multi-tenant. Dos usuarios (A, B), **el mismo rol**. La pregunta: ¿puede A tocar lo de B?

A intenta acceder a lo de B

```
1. A lista recursos              → solo las filas de A
2. A lee resource?user_id=eq.B   → array vacío o 403
3. A hace PATCH a la fila de B    → 0 filas afectadas o 403

# resultado por sonda:  VERIFIED  o  VULNERABLE
```

Cualquier VULNERABLE es territorio de bug Critical

Si A puede leer o escribir lo de B, es una fuga cross-tenant — el peor tipo de defecto. Es un finding **bloqueante**: detiene la pasada de inmediato.

// el handoff cross-layer

Un 201 No Prueba la DB

✓

La API responde 201 Created

El contrato dice "creado". Pero un POST/PATCH exitoso solo prueba que la API **aceptó** la petición.

?

¿El trigger corrió? ¿La fila quedó bien?

Los triggers pueden no ejecutarse en silencio. El total calculado, las relaciones, los timestamps — todo eso vive en la DB.

→

Baja a §DB a confirmar

Tras cada POST/PATCH exitoso, verifica persistencia y ejecución de triggers en la base. La respuesta no es la verdad.

La regla de oro de la Trifuerza

Cada capa prueba **su** verdad. La API prueba el contrato; solo la DB prueba la persistencia. Confiar en el 201 es confiar en la capa equivocada.

// quiz · smoke + UI + API

Las Capas de Arriba

Q1 El POST devuelve 201 y la UI muestra "Orden creada". ¿Es suficiente para marcar el AC como PASSED?

A Sí — 201 + UI verde = creado correctamente

B No — un 201 no prueba la DB; baja a confirmar fila + trigger

C Sí, si la consola no tiene errores rojos

No. Un 201 prueba que la API aceptó la petición, no que la DB quedó correcta — los triggers pueden no correr en silencio. Confirma persistencia (fila, total calculado, relaciones) en §DB antes de dar PASSED.

Q2 El usuario A, con su token, hace `GET resource?user_id=eq.B` y recibe las filas de B (200). ¿Veredicto?

A VULNERABLE — fuga cross-tenant, bug Critical, finding bloqueante

B VERIFIED — devolvió 200, el endpoint funciona

C Observación menor — depende del rol

VULNERABLE. A no debería ver nada de B. Leer datos de otro tenant es una fuga de aislamiento — Critical, y bloqueante: detiene la pasada de inmediato. La sonda RLS A/B es justo para cazar esto.

CAPA 03 · DB

La Capa DB
La Verdad Persistida

Confirma estado, constraints, triggers y RLS a nivel SQL. Aquí la pregunta cambia: en las sondas negativas, el **éxito es el bug**.

// después de una acción API / UI

Verificación de Estado

CHECK	PATRÓN DE QUERY
La fila existe	`SELECT * FROM {table} WHERE id = '{id}'`
La relación	`JOIN` hijo a padre, compara el conteo esperado
El trigger corrió	compara el total guardado vs `SUM(items)`
Timestamps sanos	`created_at` reciente, `updated_at >= created_at`

Aquí se cobra el 201

El "trigger corrió" es exactamente la verdad que el 201 no podía dar: si el total guardado ≠ la suma de los items, el trigger falló en silencio aunque la API respondió 201.

// la lógica invertida

Sondas de Constraints — el Éxito es el Bug

CONSTRAINT	SONDA (TODAS DEBEN ERRORAR)
FK	INSERT hijo con parent id inexistente
CHECK	UPDATE a un valor de enum inválido
UNIQUE	INSERT duplicado de columna única
NOT NULL	INSERT con NULL en columna requerida

Si la sonda tiene éxito, el constraint falta — y eso es el bug

Estas sondas deben fallar TODAS. Un INSERT inválido que pasa significa que la base no está protegiendo la integridad. **Envuelve cada sonda en una transacción y haz ROLLBACK** para no contaminar staging.

// barridos de integridad

Cazar Datos Corruptos

SQL · estas queries deben volver 0 filas

```sql
-- huérfanos: hijos sin padre
SELECT c.* FROM child c LEFT JOIN parent p ON c.fk=p.id WHERE p.id IS NULL;

-- calc mismatch: total guardado ≠ suma real
SELECT o.id, o.total, COALESCE(SUM(i.qty*i.unit_price),0) AS calc
FROM orders o LEFT JOIN order_items i ON o.id=i.order_id
GROUP BY o.id, o.total HAVING o.total != calc;

-- estados de dominio imposibles
SELECT * FROM orders WHERE status='delivered' AND payment_status!='completed';
```

Cualquier fila devuelta es un defecto de integridad

Un pedido "entregado" sin pago "completado" es un estado que el negocio jamás debería permitir. Estos barridos cazan la corrupción que ninguna prueba de UI vería.

// RLS real + cascadas

RLS a Nivel SQL y Limpieza

RLS de verdad

El rol `qa_team` suele saltar RLS. Para probarla real, impersona el JWT:

```
BEGIN;
SET LOCAL request.jwt.claim.sub = '{user-a}';
SELECT * FROM {table}; -- solo filas de A
ROLLBACK;
```

Cascadas de limpieza

Tras probar flujos CRUD, verifica que la limpieza cascadea:

- DELETE padre → ¿los hijos se van?
- Soft-delete: ¿flag puesto, fila aún presente?
- Registra `CLEANUP COMPLETE / INCOMPLETE`

Si el SQL directo salta la RLS

Cae a la sonda API §RLS A/B — esa pasa por el enforcement real de la aplicación. Dos caminos para la misma verdad de seguridad.

// quiz · capa DB

La Verdad Persistida

Q1 Sondas la constraint UNIQUE insertando un duplicado, y el INSERT **tiene éxito** (no errora). ¿Qué significa?

A Bien — la base aceptó el dato, todo en orden

B Bug — el constraint UNIQUE falta; la sonda negativa debía errorar

C Observación — depende de la columna

Bug. En sondas de constraints la lógica se invierte: el éxito es el defecto. Un duplicado que pasa = la base no protege la unicidad. Por eso las sondas FK/CHECK/UNIQUE/NOT NULL deben errorar todas (en transacción + ROLLBACK).

Q2 Tu barrido devuelve un pedido con `status='delivered'` y `payment_status='pending'`. ¿Acción?

A Ignorar — la UI no lo muestra, no afecta al usuario

B Defecto de integridad — un estado de dominio imposible que el negocio nunca debe permitir

C Normal — los estados son independientes

Defecto de integridad. "Entregado sin pago completado" es un estado imposible. Los barridos de integridad cazan exactamente esta corrupción silenciosa — invisible en la UI, real en los datos.

EL OFICIO

La Disciplina Que Hace la Pasada Confiable

Cuándo detenerse y cuándo seguir, cómo tratar el tiempo, y por qué ningún test puede quedar en NOT RUN.

// la pausa graduada

Bloqueante vs No-Bloqueante

Un FAIL durante el testing de la feature **no** es automáticamente Critical y **no** detiene la pasada por sí solo. Triagea primero, luego decide.

Bloqueante → STOP

- smoke / ambiente caído
- corrupción / pérdida de integridad
- seguridad explotable (RLS `VULNERABLE`, bypass de auth, cross-tenant)

vs

No-bloqueante → log + sigue

- cosmético, gap de validación menor
- edge case en un TC no crítico
- finding de seguridad/framework-default pendiente de recalibración

Por qué no parar por todo

Pausar una pasada de 17 TCs por un hallazgo cosmético desperdicia el dispatch y pierde cobertura. Un finding no-bloqueante se registra y la pasada **continúa**; todos se surfacean juntos al cierre de Stage 2. Un bloqueante genuino sí detiene de inmediato.

// probar el tiempo sin esperar

Casos Dependientes del Tiempo (TTL)

Magic links, OTPs, tokens, sesiones, caches: "a las 14:59 funciona, a las 15:01 no". No se prueban haciendo clic más rápido — necesitas controlar el tiempo. Elige la opción más alta que el stack permita.

1

Fixture / TTL corto

La mejor. Si el env expone un TTL configurable, ponlo en segundos y prueba el borde 14:59/15:01 real. Ejercita el código de expiry verdadero.

2

Clock-mock

Si el stack permite inyectar/congelar el tiempo, avanza el reloj y asegura la expiración — siempre que el mock maneje la lógica real, no una rama de test.

3

Defer manual escrito

Último recurso. Documenta el borde que no pudiste ejercitar, por qué, y el follow-up concreto (ej. "añadir `MAGIC_LINK_TTL` override").

Nunca un `BLOCKED` a secas

Marca `BLOCKED — needs time fixture` con la opción elegida (1/2/3) y una línea de razón. Un BLOCKED sin camino de decisión pierde cobertura en silencio.

// cerrar la pasada con rigor

Sin NOT RUN · Evidencia que Sirve

Propagación de estado

Al cerrar Stage 2, cada outline / TC tiene **PASSED o FAILED**. Sin NOT RUN. Si uno queda NOT RUN: o está fuera de scope (di por qué en Notes) o está incompleto (vuelve al loop).

Disciplina de evidencia

- Nombre: `{KEY}-{n}-ac{N}-desc.png`
- Captura en el **estado de fallo**, no navegando hacia él
- `evidence/` gitignored, nunca commiteado

El log es el handoff a Stage 3

El bloque Stage-2 de `test-session-memory.md` (smoke + tablas UI/API/DB + Findings) es lo que Stage 3 consume: los Findings alimentan los bugs, los pass/fail alimentan el ATR.

// quiz · disciplina

El Oficio de la Pasada

Q1 En el TC #4 de 17 encuentras un texto desalineado (cosmético). ¿Detienes la pasada?

A Sí — todo FAIL detiene la pasada y se reporta de inmediato

B No — es no-bloqueante: lo registro y sigo; lo surfaceo todo junto al cierre

C Sí — abro un bug Critical para no olvidarlo

No, sigue. Un cosmético es no-bloqueante: se registra en Findings, el TC queda FAILED y la pasada continúa para no perder los otros 13 TCs. Solo lo bloqueante (env/corrupción/seguridad) detiene. Un FAIL no es auto-Critical.

Q2 Un TC valida que un magic link expira a los 15 min, pero no hay forma de adelantar el reloj hoy. ¿Cómo lo cierras?

A Lo dejas en `NOT RUN`

B `BLOCKED` a secas — no se pudo correr

C `BLOCKED — needs time fixture` + la opción elegida y la razón

BLOCKED — needs time fixture. Nunca un BLOCKED desnudo ni un NOT RUN. Documenta el borde, por qué no se pudo, y el follow-up (opción 3: defer escrito). Así la cobertura no se pierde en silencio y viaja a Stage 3.

// todo junto

La Pasada Completa de Testing de una Feature

Smoke ✓ → UI → API → DB → Findings

- **Smoke primero** — Go/No-Go del ambiente, ⩽10 min, siempre revisa la red.
- **Capas por tipo de feature** — el orden lo decide qué cambió; cada capa prueba su verdad.
- **Triagea cada finding** — bloqueante detiene, no-bloqueante registra y sigue.
- **Cierra con rigor** — sin NOT RUN, evidencia en el estado de fallo, Findings al handoff de Stage 3.

// el checklist del tester

Cómo Hacerlo Bien

- **Smoke antes que todo** — un ambiente roto da bugs falsos.
- **Desconfía de la UI** — revisa la red aunque se vea verde.
- **Lee los status codes como evidencia** — cada uno prueba algo.
- **Corre la sonda RLS A/B** — multi-tenant siempre.
- **Confirma en DB tras cada write** — el 201 no basta.
- **Sondas negativas: éxito = bug** — en transacción + ROLLBACK.
- **Prueba las transiciones inválidas** — ahí viven los defectos.
- **Cierra sin NOT RUN** — cada TC PASSED o FAILED, con evidencia.

La meta

Que ningún defecto cruce las tres capas sin ser visto: ni el que la UI esconde, ni el que la API maquilla, ni el que la DB corrompe en silencio.

// para llevar

Tres Ideas que se Quedan

1

Cada capa, su verdad

La UI muestra, la API contrata, la DB persiste. Un 201 no prueba la base; una pantalla verde no prueba la API.

2

El negativo es el oro

Los status de error, las transiciones inválidas, las sondas de constraint (éxito=bug) y RLS A/B cazan lo que el happy path nunca toca.

3

Disciplina sobre impulso

Smoke primero, triage graduado, sin NOT RUN. La cobertura completa vale más que parar por el primer hallazgo.

Tres Capas. Una Verdad. Ningún Punto Ciego.

Smoke valida el ambiente → la Trifuerza valida la feature: UI (lo que se ve), API (el contrato), DB (lo que persiste). Prueba el negativo, confirma una capa abajo, cierra sin NOT RUN.

← → navegar · S notas del orador · O resumen · T tema