Testing Efectivo en Java

Círculo Siete Capacitación

Clase 11 de 12

30 Agosto 2025

El contract testing (pruebas de contrato)

El contract testing

- Es una práctica de testing utilizada principalmente en arquitecturas distribuidas (por ejemplo, microservicios), cuyo objetivo es verificar que dos sistemas que se comunican entre sí (generalmente un proveedor y un consumidor) cumplen con un contrato previamente establecido.
- Ese contrato describe cómo deben interactuar: qué datos se envían, en qué formato, qué endpoints están disponibles, qué respuestas se esperan, códigos de estado, headers, etc.

Idea principal

- En vez de depender solo de pruebas de integración end-to-end (lentas y frágiles).
- El contract testing valida que los consumidores y proveedores estén alineados en sus expectativas.
- Asegurando que los cambios en un servicio no rompan a otros.

Roles

- Proveedor (Provider)
  - El servicio que expone una API (ejemplo: un servicio de facturación).
- Consumidor (Consumer)
  - El cliente que usa esa API (ejemplo: un servicio de pedidos que consulta facturación).

Ejemplo simple

- El consumidor espera que el endpoint /clientes/{id} devuelva un JSON con id, nombre y email.
- El contrato define esa expectativa.
- El proveedor ejecuta las pruebas de contrato para verificar que su implementación cumple lo acordado.

Beneficios

- Prevención temprana de errores: detecta inconsistencias antes de desplegar.
- Menor acoplamiento: los equipos pueden trabajar en paralelo.
- Mayor confianza: cada cambio puede validarse sin necesidad de desplegar todo el sistema.
- Automatización: se integra fácilmente en pipelines de CI/CD.

Herramientas comunes

- Pact (muy popular, especialmente con microservicios y APIs REST/HTTP).
- Spring Cloud Contract (en el ecosistema de Spring).
- Postman Contract Testing (con OpenAPI/Swagger).

Qué valida exactamente

- Estructura: campos obligatorios/opcionales, tipos, formatos (ej. email, uuid, iso-8601).
- Semántica: valores permitidos, relaciones entre campos, defaults.
- Comportamiento: códigos HTTP, headers, reglas de negocio, error shapes.
- Evolución: compatibilidad hacia atrás al cambiar contratos.

Estilos de contratos

- Consumer-Driven Contracts (CDC)
  - Los consumidores definen expectativas mínimas; el proveedor las verifica. Favorece independencia y evita APIs sobredimensionadas.
- Provider-Driven / Especificación central (OpenAPI/AsyncAPI)
  - Un contrato fuente de verdad publicado por el proveedor; los consumidores generan clientes y tests.
- En la práctica, muchos equipos combinan CDC (Pact/SSC) + OpenAPI (documentación/descubrimiento).

Flujo típico (CDC con Pact)

- Consumidor escribe tests contra un mock provider → genera un pact file.
- Publica el pact en un Broker (ej. Pact Broker).
- Proveedor ejecuta provider verification contra ese pact (con estados/fixtures).
- El Broker marca la compatibilidad (matrices por versión/branch/ambiente).
- CI/CD bloquea despliegues si no hay verificación compatible.

Flujo típico
(Provider-first con Spring Cloud Contract)

- Proveedor define contratos (DSL Groovy/YAML) + stubs.
- Stub Runner publica artefactos (JAR) con stubs para que los consumidores testeen sin tocar al proveedor real.
- El build del proveedor falla si la implementación rompe el contrato.

Dónde encaja en la pirámide de tests

- Va encima de unit/integration locales y debajo de end-to-end.
- Objetivo: reducir la cantidad de E2E frágiles, cubriendo la compatibilidad con contratos rápidos y deterministas.

Buenas prácticas (lo que más paga)

- Contratos pequeños y centrados en casos de uso (no “todo el OpenAPI de una”).
- Schemas reutilizables con restricciones claras (ej. JSON Schema + pattern, minLength, format).
- Versionado explícito de contratos (semver) y deprecaciones con ventanas acordadas.
- States en provider tests para datos consistentes (fixtures idempotentes).
- Naming estable para endpoints/fields; agrega campos nuevos como opcionales primero.
- Matrices de compatibilidad por versión (consumer x provider) en el broker.
- Contratos para errores (no solo 200) con payloads bien definidos.
- Contratos para idempotencia (ej. Idempotency-Key) y ordenamiento/paginación (tipos, límites).
- Automatiza el “release gate”: no se despliega si el contrato no está verificado en ese commit.

Antipatrones (lo que rompe)

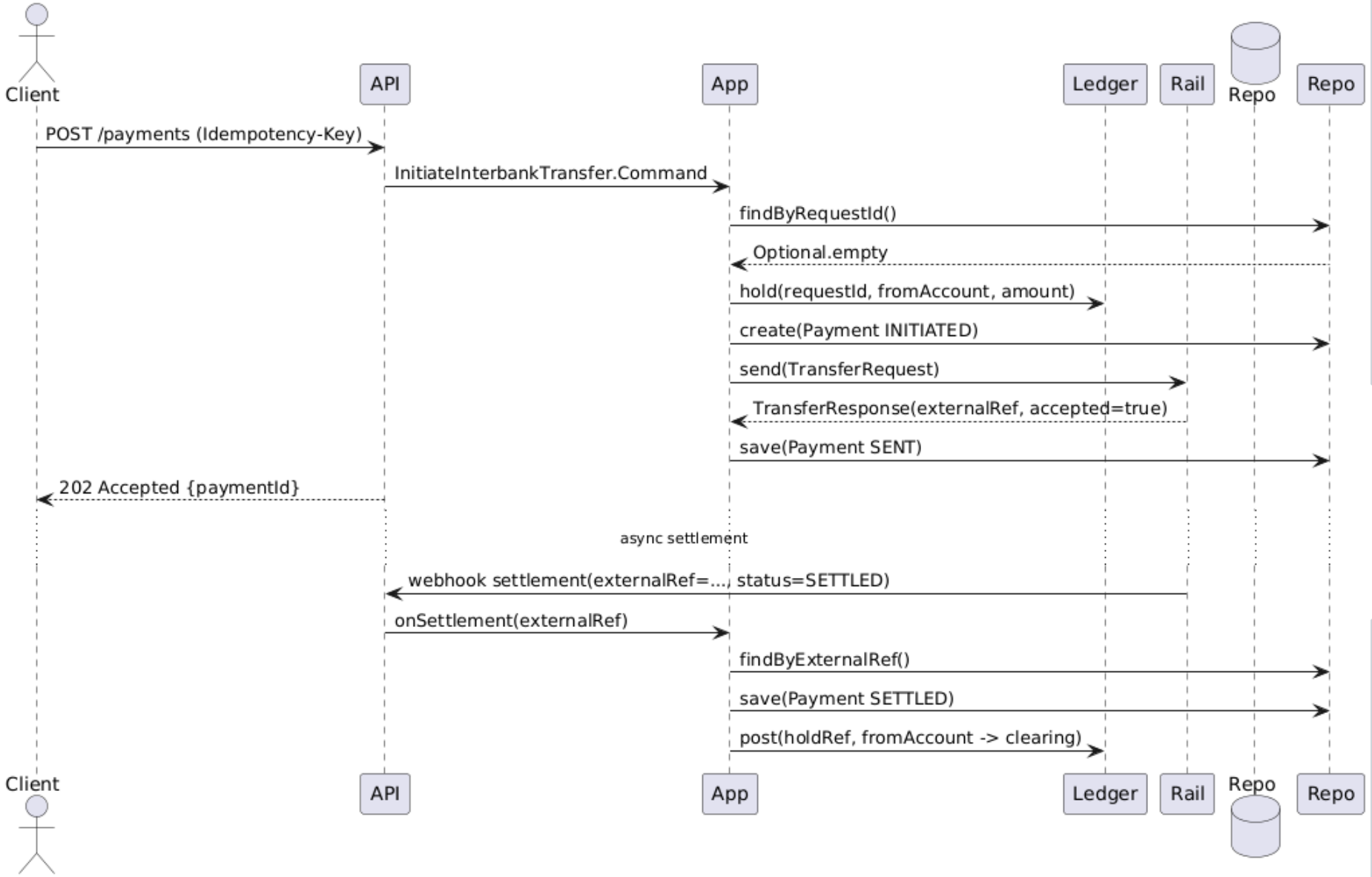
- Contratos que duplican toda la API (ruido, fricción).
- Mocks demasiado rígidos (matchers exactos para valores variables; usa matchers por tipo/patrón).
- Tests de contrato que insertan lógica de negocio compleja → mantenlos en unit/integration.
- No versionar → rompe consumidores silenciosamente.
- Sólo casos felices; sin 4xx/5xx ni límites (paginación, pageSize máximos).

Cambios y compatibilidad

- Compatible: agregar un campo opcional, aceptar valores extra, añadir endpoint nuevo.
- Potencialmente rompedor: renombrar/quitar campo, endurecer validaciones, cambiar tipos/formatos, cambiar semántica de errores.
- Estrategia: expand-then-contract → añade opcional, migra consumidores, luego remueve.

Laboratorio

- Revisar el código


Client
API
App
Ledger
Rail
Repo
Repo
POST /payments (Idempotency-Key)
InitiateInterbankTransfer.Command
findByRequestId()
Optional.empty
hold(requestId, fromAccount, amount)
create(Payment INITIATED)
send(TransferRequest)
TransferResponse(externalRef, accepted=true)
save(Payment SENT)
202 Accepted {paymentId}
async settlement
webhook settlement(externalRef=..., status=SETTLED)
onSettlement(externalRef)
findByExternalRef()
save(Payment SETTLED)
post(holdRef, fromAccount -> clearing)
Client
API
App
Ledger
Rail
Repo
Repo

Tips finales

- Mantén contratos pequeños y específicos (un happy path por interacción + errores relevantes: payload inválido, externalRef desconocido → 400).
- Versiona los pacts en Git y/o publícalos en Pact Broker para gobernanza entre equipos.