

goldbergonyi / javascript-testing-best-practices Public

[Code](#)[Issues 28](#)[Pull requests 6](#)[Actions](#)[Projects](#)[Wiki](#)[Security](#)[Insights](#)[master](#)

...

javascript-testing-best-practices / readme-es.md

[goldbergonyi Merge pull request #164 from jfernandezpe/minor-spanish-changes ...](#)[5 contributors](#)[1990 lines \(1265 sloc\)](#)

...

JavaScript & Node.js Testing Best Practices

Test Structure



Backend



Frontend



CI & More



Por qué esta guía puede hacerte llevar tus habilidades de testing al siguiente nivel



46+ buenas prácticas: Súper comprensiva y exhaustiva

Esta es una guía completa para JavaScript y Node.js de la A a la Z. Resume y selecciona docenas de los mejores post de blogs, libros, y herramientas ofrecidas en el mercado



Avanzado: Va 10.000 kilómetros más allá de lo básico

Súbete a un viaje que va más allá de lo básico, llegando a temas avanzados como testeando en producción, mutation testing, property-based testing y muchas otras herramientas estratégicas y profesionales. Si lees esta guía completamente es probable que tus habilidades de testing acaben por encima de la media



Full-stack: front, backend, CI, cualquier cosa

Empieza por comprender las técnicas de testing ubicuas que son la base de cualquier nivel de aplicación. Luego, profundiza en tu área de elección: frontend/UI, backend, CI o tal vez todos

Escrita por Yoni Goldberg

- Consultor JavaScript & Node.js
- [Testing Node.js & JavaScript de la A a la Z](#) - Mi curso completamente online con más de [10 horas de video](#), 14 tipos de test y más de 40 buenas prácticas
- [Sígueme en Twitter](#)

Traducciones - leelas en tu propio idioma

- [CNChino](#) - cortesía de [Yves yao](#)
- [KRCoreano](#) - cortesía de [Rain Byun](#)
- [PLPolaco](#) - cortesía de [Michał Biesiada](#)
- [ESEspañol](#) - cortesía de [Miguel G. Sanguino](#)
- [BRPortugués-BR](#) - cortesía de [Iago Angelim Costa Cavalcante](#), [Douglas Mariano Valero](#) y [koooge](#)
- ¿Quieres traducir a tu propio lenguaje? por favor abre una issue

Tabla de Contenidos

Sección 0: La Regla de Oro

Un solo consejo que inspira a todos los demás (1 apartado especial)

Sección 1: La Anatomía de un Test

La base - estructurando test claros (12 apartados)

Sección 2: Backend

Escribiendo test de backend y microservicios eficientemente (8 apartados)

Sección 3: Frontend

Escribiendo test para web UI incluyendo test de componente y E2E (11 apartados)

Sección 4: Midiendo la Efectividad de los Test

Vigilando al vigilante - midiendo la calidad de los test (4 apartados)

Sección 5: Integración Continua

Pautas para Integración Continua en el mundo de JS (9 apartados)

Sección 0 : La Regla de Oro

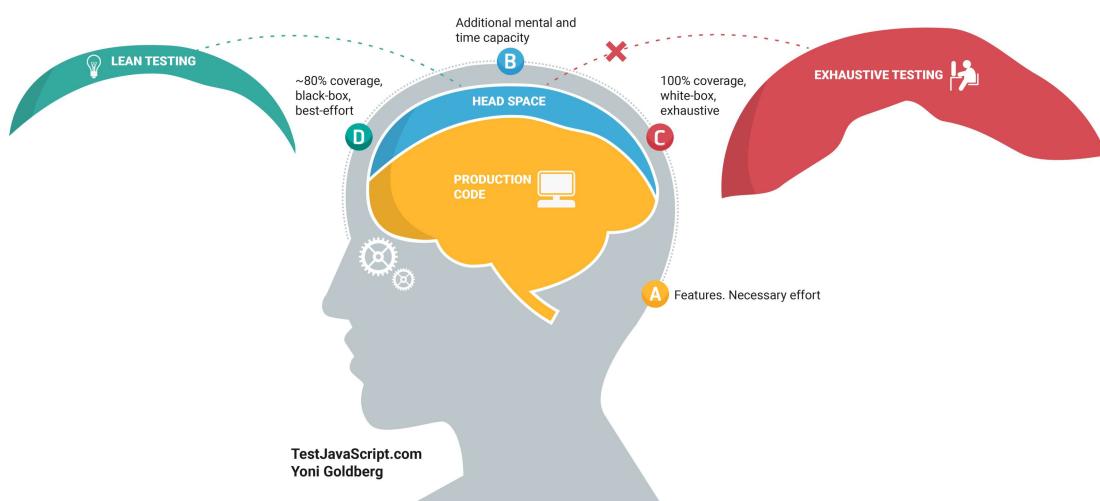
0 La Regla de Oro: Diseñando testing ligero

 **Haz:** El código de los test no es como el código de producción - diséñalo para que sea simple, corto, sin abstracciones, plano, agradable de trabajar, ligero. Uno debe mirar un test y entender la intención al instante

Nuestras mentes están llenas por el código de producción, no tenemos espacio de cabeza para añadir más cosas complejas. Si intentamos que introducir otro código desafiante en nuestro cerebro, ralentizará todo el equipo, lo que va en contra de la razón por la que hacemos testing. Prácticamente esta una de las razones por la que muchos equipos simplemente abandonan el testing

Los test son una oportunidad de tener algo más: un asistente amable y soniente, con el que es un placer trabajar y da mucho valor a cambio de una inversión muy pequeña. La ciencia nos dice que tenemos dos sistemas cerebrales: el sistema 1 se usa para actividades sin esfuerzo como conducir un automóvil en una carretera vacía y el sistema 2, que está destinado a operaciones complejas y conscientes como resolver una ecuación matemática. Diseña tus test para el sistema 1, cuando observes el código de un test, debería parecer tan fácil como modificar un documento HTML y no como resolver $2X(17 \times 24)$

Esto se puede lograr mediante la selección de técnicas cuidadosamente, herramientas y objetivos de test que son rentables y proporcionan un gran retorno de la inversión. Testea solo lo que sea necesario, esfuérzate por mantenerlo ágil, a veces incluso vale la pena abandonar algunos test y cambiar la confiabilidad por agilidad y simplicidad



La mayoría de los siguientes consejos son derivados de este principio

¿Listo para empezar?

Sección 1: La Anatomía de un Test



1.1 Incluye 3 partes en los nombres de tus test

Haz: El reporte de un test debe indicar si la revisión de la aplicación actual cumple los requisitos para las personas que no están necesariamente familiarizadas con el código: el tester, el DevOps que está desplegándolo y el futuro tú de dentro de dos años. Esto se puede lograr si los test hablan al nivel de los requisitos e incluyen 3 partes:

- (1) ¿Qué se está testeando? Por ejemplo, el método `ProductsService.addNewProduct`
- (2) ¿Bajo qué escenario y circunstancias? Por ejemplo, no se pasa ningún precio al método
- (3) ¿Cuál es el resultado esperado? Por ejemplo, el nuevo producto no está aprobado

De lo contrario: Un despliegue falla, un test llamado "Aregar producto" ha fallado. ¿Esto te dice exactamente qué está funcionando mal?



Nota: Cada apartado tiene ejemplos de código y, en ocasiones, también una imagen ilustrativa. Haga clic para ampliar

► **Código de Ejemplo**

► **Créditos y más información**



1.2 Estructura tus test con el patron AAA

Haz: Estructura tus test con 3 secciones bien separadas Ajustar, Actuar y Afirmer (AAA en inglés Arrange, Act & Assert). Seguir esta estructura garantiza que quien lea nuestro test no se estruje el cerebro en comprender los test:

1a A - Ajustar: configura el código, crea el escenario que el test pretende simular. Esto podría incluir crear instancias de la unidad bajo el constructor del test, agregar registros de base de datos, mocks y stubs de objetos y cualquier otro código necesario

2a A - Actuar: Ejecuta la unidad en test. Normalmente 1 línea de código

3a A - Afirmer: Comprobar que el valor recibido satisface las expectativas. Normalmente 1 línea de código

✗ **De lo contrario:** No solo emplearas horas comprendiendo el código principal, si no que lo que debería haber sido la parte más simple del día (testing) te ha estrujado el cerebro

►  Código de Ejemplo

○ 1.3 Describe las expectativas en el lenguaje del producto: usa aserciones estilo BDD

✓ **Haz:** Escribir el código de tus test de forma declarativa permite que aquel que lo lea tenga al instante la idea sin tener que estrujarse el cerebro. Cuando escribes el código de los test de forma imperativa estará lleno de condiciones lógicas y obligas al que lo lee a gastar mucho más tiempo en comprenderlo. En este caso, escribe el código de la forma más humana, de forma declarativa con BDD, usando `expect` o `should` y sin usar código personalizado. Si Chai o Jest no incluyen las aserciones que deseas y se hace muy repetitivo, siempre puedes [extender Jest con Jest matcher](#) o escribir un [plugin de Chai](#)

✗ **De lo contrario:** El equipo escribirá menos test y acabará marcando los test más molestos con `.skip()`

►  Código de Ejemplo

○ 1.4 Acercarse al testing caja-negra: Testea solo métodos públicos

 **Haz:** Testear las partes internas suele traer un gasto extra enorme para obtener muy poco beneficio. Si tu código/API comprueba todos los resultados posibles correctamente, ¿deberías perder las próximas 3 horas en comprobar como esta funcionando internamente y después mantener esos test tan frágiles? Cada vez que se comprueba un comportamiento público, la implementación privada es implícitamente testeada y tus test se romperán sólo si hay un problema concreto (por ejemplo una salida incorrecta). Este enfoque también es conocido como `behavioral testing` (testing de comportamiento). Por otro lado, si se testean las partes internas (caja-blanca) tu enfoque cambia de planificar la salida del componente a detalles minúsculos, y tus test pueden romperse debido a refactors de código menores sin que se rompan los test de salida, por lo que aumenta tremadamente el mantenimiento de los mismos

 **De lo contrario:** Tus test se comportaran como la fabula de [que viene el lobo](#): gritando falsos positivos (por ejemplo, un test falla porque se cambio el nombre a una variable probada). Como es de esperar, la gente empezara a ignorar estos test hasta que un día ignoren un test de verdad...

►  Código de Ejemplo

○ 1.5 Eligiendo los dobles de los test correctamente: Evita mocks en favor de stubs y spies

 **Haz:** Los dobles de los test son un mal necesario porque están acoplados a las tripas de la aplicación, sin embargo, algunos proporcionan un valor inmenso ([Aquí tienes un bueno recordatorio de que son los dobles de los test: mocks vs stubs vs spies](#))

Antes de usar un doble, hazte una simple pregunta: ¿Lo voy a usar para testear una funcionalidad que aparece, o puede aparecer, en el documento de requisitos? Si no, eso huele que estas testeando partes privadas

Por ejemplo, si quieres testear que tu app se comporta razonablemente cuando el servicio de pagos está caído, deberías hacer un stub del servicio de pagos y devolver un 'Sin respuesta' para asegurar que la unidad que está siendo testeada devuelve el valor correcto. Esto verifica que el comportamiento/respuesta/resultado de nuestra app en ciertos escenarios. También podrías usar un spy para asegurar que un email ha sido enviado cuando este servicio está caído — esto es nuevamente una verificación de comportamiento que probablemente aparezca en el documento de requisitos ("Enviar un correo electrónico si no se pudo guardar el pago"). En el lado opuesto, si se mockea el servicio de pagos y se asegura que se haya llamado con el tipado correcto— entonces tu test esta comprobando cosas internas que no tienen nada que ver con la funcionalidad de la app y es muy probable que cambien con frecuencia

 **De lo contrario:** Cualquier refactor de código te exigirá buscar todos los mocks que tengas y tendrás que actualizarlos en consecuencia. Los test pasan de ser un amigo útil a una carga más

►  Código de Ejemplo



¿Quieres aprender todas esto con video en directo?

Visita el curso online [Testing Node.js & JavaScript From A To Z](#)



1.6 No uses "foo", usa datos realistas

 **Haz:** A menudo, los bugs de producción se revelan bajo una entrada muy específica y sorprendente: cuanto más realista sea la entrada de un test, mayores serán las posibilidades de detectar bugs temprano. Utiliza librerías dedicadas como [Faker] (<https://www.npmjs.com/package/faker>) para generar datos pseudo-reales que se asemejan en variedad y forma a los datos de producción. Por ejemplo, dichas librerías pueden generar números de teléfono realistas, nombres de usuario, tarjetas de crédito, nombres de empresas e incluso texto "lorem ipsum". También puedes crear algunos test (además de los test unitarios, no como un reemplazo) que aleatorizan los datos falsos para forzar la unidad que estamos testeando o incluso importar datos reales de su entorno de producción. ¿Quieres llevarlo al siguiente nivel? Ve la próxima sección (test basados en propiedades)

 **De lo contrario:** Todo tus test de desarrollo estarán en verde falsamente cuando uses datos sintéticos como "Foo", pero luego en producción pueden ponerse en rojo cuando un hacker use cadenas extrañas como "@3e2ddsf . ##' 1 fdsfds . fds432 AAAA"

►  Código de Ejemplo

○ 1.7 Testea muchas combinaciones de entrada utilizando test basados en propiedades

 **Haz:** Por lo general elegimos pocos datos de entrada por cada test. Incluso cuando el formato de entrada se parece a datos reales (ver sección "no uses foo"), cubrimos solo unas pocas combinaciones de datos de entrada (method("", true, 1), method("string", false, 0)). Sin embargo, en producción, un API que es llamada con 5 parámetros puede ser invocada por miles de combinaciones diferentes, una sola puede hacer que nuestro proceso falle ([ver Fuzz Testing](#)). ¿Qué tal si pudieras escribir un solo test que envíe 1000 combinaciones de diferentes entradas automáticamente y capture qué entrada hace que código no devuelva la respuesta correcta? Los test basados en propiedades son una técnica que hace exactamente eso: al enviar todas las combinaciones de entrada posibles a la unidad que está siendo testada, aumenta la probabilidad de encontrar un bug. Por ejemplo, dado un método —addNewProduct(id, name, isDiscount)— las librerías compatibles llamarán a ese método con muchas combinaciones (números, textos y booleanos) como (1, "iPhone", false), (2, "Galaxy", true). Puedes ejecutar test basados en propiedades usando tu librería de test favorita (Mocha, Jest, etc) como [js-verify](#) o [testcheck](#) (mucho mejor documentada). Actualizado: Nicolas Dubien sugiere en los comentarios [checkout fast-check](#) que parece ofrecer características adicionales y es activamente mantenida

 **De lo contrario:** Inconscientemente, eliges los datos de entrada para tus test que cubren solo las ramas de código que funcionan bien. Desafortunadamente, esto disminuye la eficiencia de los test como vehículo para detectar bugs

►  Código de Ejemplo

○ 1.8 Si lo necesitas, usa solo snapshots cortos y en el propio test

 **Haz:** Cuando hay necesidad de usar [snapshot testing](#), usa solo snapshots cortos y bien enfocados (por ejemplo 3-7 lineas) y que estén incluidos en el propio test ([Inline Snapshot](#)) y no como ficheros externos. Mantener esta dirección te garantiza que tus test se explican por si mismos y a la vez que sean menos frágiles

Por otro lado, los tutoriales y herramientas basados en 'classic snapshots' tienden a guardar ficheros muy grandes en medios externos (por ejemplo component rendering markup, API JSON result) cada vez que se ejecutan los test para comparar los resultados recibidos con la versión guardada. Esto, por ejemplo, puede asociar nuestro test a 1000 lineas con 3000 valores de datos que quien este escribiendo test jamás leerá ni razonará. ¿Por qué está mal esto? Al hacerlo, hay 1000 razones para que tu test falle - tan solo el cambio de una linea de código es suficiente para que el snapshot se invalide y es muy probable que esto ocurra a menudo. ¿Como de frecuente? cada espacio, comentario o pequeño cambio de css/html. Y no solo eso, el nombre del test no nos va a dar ni una sola pista de que está fallando, solo verifica que esas 1000 lineas han cambiado. Además obliga a quien escribe los test a asumir como correcto un fichero enorme que no ha podido inspeccionar y corroborar. Todo estos son síntomas de un test oscuro que no está bien enfocado y trata de cubrir demasiadas cosas a la vez

Vale la pena señalar que hay algunos casos en los que los snapshots grandes y externos son buenos - cuando comprobamos el esquema y no los datos (ignorando los valores y centrándonos en los campos) o en los casos en el que el documento no va a cambiar apenas en el tiempo

 **De lo contrario:** Un test UI falla. El código parece correcto, la pantalla esta pintando todos los pixels correctamente, ¿que ha pasado? tu test de snapshot ha encontrado una diferencia entre el origen y lo que ha recibido al ejecutarse: simplemente hay un espacio añadido en cualquier parte...

►  Código de Ejemplo

○ 1.9 Evitar fixtures globales y seeds, añade datos por cada test

 **Haz:** Siguiendo la regla de oro (sección 0), cada test debe añadir y actuar en su propio conjunto de filas en base de datos para evitar el acoplamiento y poder explicar fácilmente el flujo del test. En realidad muchos testers se saltan esta regla al añadir datos a la DB solo una vez antes de ejecutar los test ([también conocido como 'test fixture'](#)) a favor de mejorar el rendimiento. Si bien el rendimiento es una preocupación válida — puede mitigarse de otras formas (consulte la sección "test de componentes"), sin embargo, la complejidad del test es más dolorosa que otras consideraciones la mayoría de las veces. De manera práctica, haz que cada test añada explícitamente los registros que necesitas y actúe sobre ellos. Si el rendimiento se convierte en algo crítico — se puede llegar al compromiso de utilizar los mismos datos en un conjunto de test siempre que no se muten los datos (por ejemplo en queries)

 **De lo contrario:** Muchos test fallaran, un despliegue se aborta, nuestro equipo perderá mucho tiempo, ¿tenemos un bug? vamos a investigar, ah no — parece que dos test están mutando el mismo dato

►  Código de Ejemplo

1.10 No captures errores, esperalos

 **Haz:** Cuando queremos comprobar que una entrada lanza un error, nos puede parecer correcto usar try-catch-finally y asertar que se entra por el catch. El resultado es un test incomodo y verboso (ejemplo a continuación) que nos oculta la intención de un test muy simple y las expectativas del resultado

Una alternativa más elegante sería usar solo la aserción de una sola linea que tiene Chai: `expect(method).to.throw` (o en Jest: `expect(method).toThrow()`). Es totalmente obligatorio también asegurarse de que la excepción contenga una propiedad que indique el tipo de error, de lo contrario, lanzando solo un error genérico, la app no podrá hacer mucho más que mostrarle un error decepcionante para el usuario

 **De lo contrario:** Sería muy difícil deducir a partir de los reportes de test (por ejemplo, reporte de CI) que es lo que ha salido mal

►  Código de Ejemplo



1.11 Etiqueta tus test

Haz: Deben ejecutarse diferentes test en diferentes escenarios: quick smoke, IO-less, los test deben ejecutarse cuando un desarrollador guarda o hace commit de un fichero, los test end-to-end suelen ejecutarse cuando un nuevo pull request es añadido, etc. Esto se puede lograr etiquetando los test con tags como #cold #api #sanity para que se pueda filtrar e invocar solo el subconjunto deseado. Por ejemplo, así es como se ejecutan solo el grupo sanity test con Mocha: mocha—grep 'sanity'

De lo contrario: Ejecutar todos los test, incluidos los test que realizan docenas de queries a base de datos, cada vez que un desarrollador hace un pequeño cambio, puede ser extremadamente lento y provocar que los desarrolladores ignoren correr los test

► Código de Ejemplo

1.12 Categoriza los test en al menos 2 niveles

Haz: Aplica cierta estructura a tu conjunto de test para que un visitante ocasional pueda comprender fácilmente los requisitos (los test siempre son la mejor documentación) y los diversos escenarios que estamos testeando. Una práctica común para esto es crear al menos 2 bloques 'describe' antes de tus test: el primero es para el nombre de la unidad que está siendo testeada y el segundo es para un nivel adicional de categorización como el escenario o las categorías personalizadas (ver ejemplos de código y pantallazos más abajo). Hacerlo también mejorará los reportes de los test: quien los lea deducirá fácilmente las categorías de los test, profundizará en aquellas que lo desee y podrá relacionar mejor los test fallidos. Además, será mucho más fácil para un desarrollador navegar a través del código de un conjunto de test amplio. Existen múltiples formas de estructurar tus test que deben ser consideradas, como [given-when-then](#) y [RITE](#)

De lo contrario: Cuando nos enfrentamos a un reporte con una lista plana de test, tendremos que leer rápidamente textos largos para determinar los escenarios principales y relacionar los test fallidos. Considera el siguiente caso: Cuando 7/100 test fallan, revisar una lista plana te exige leer el texto de los test que fallan para ver como se relacionan entre ellos y que tienen en común. Sin embargo, en un reporte jerarquizado, si los 7 están bajo un mismo flujo o categoría, puedes saber rápidamente cual o donde puede estar la causa raíz del fallo

►  Código de Ejemplo

○ 1.13 Otras buenas prácticas genéricas sobre higiene de los test

 **Haz:** Esta publicación se centra en consejos de test relacionados con, o al menos, que se pueden exemplificar en Node JS. Sin embargo, esta sección agrupa algunos consejos no relacionados con Node que son bien conocidos

Aprenda y practique [principios TDD](#)—son extremadamente valiosos para muchos pero no te dejes intimidar si no se ajustan a tu estilo, no eres el único. Considera escribir los test antes que el código con el [estilo rojo-verde-refactor](#), te asegura que cada test chequea exactamente una cosa, cuando encuentras un bug—antes de corregirlo escribe un test que lo detecte como bug en el futuro, dejando que cada test falle al menos una vez antes de convertirlo en un verde, comienza el inicio de un modulo escribiendo código muy simple y rápidamente, que satisfaga el test, luego lo refatorizamos gradualmente hasta que nuestro código tenga el nivel deseado en producción, evitando siempre cualquier dependencia con el entorno (rutas en disco, sistema operativo, etc)

 **De lo contrario:** Echarás de menos las perlas de sabiduría que se han ido recolectando durante décadas

Sección 2: Backend Testing

○ 2.1 Enriquece tu abanico de test: mira más allá de los test unitarios y la pirámide

 **Haz:** La [pirámide de test](#), con 10+ años de antigüedad, es un modelo excelente y relevante que sugiere tres tipos de test e influye en la estrategia de testeo de la mayoría de los desarrolladores. Al mismo tiempo, surgieron un puñado de nuevas y brillantes técnicas de testeo que se esconden en las sombras de la pirámide de test. Dados todos los cambios que hemos visto en los últimos 10 años (microservicios, cloud, serverless), ¿es posible que un modelo algo antiguo se adapte a *todos* los tipos de aplicaciones? ¿No debería el mundo del testing considerar aceptar nuevas técnicas?

No me malinterpretes, en 2019 la pirámide de test, el TDD y los test unitarios siguen siendo una técnica buena y probablemente sean la mejor combinación para muchas aplicaciones. Sólo como cualquier otro modelo, a pesar de su utilidad, [a veces debe estar equivocado] (https://en.wikipedia.org/wiki/All_models_are_wrong). Por ejemplo, considera una aplicación IOT que ingiere muchos eventos en un bus de mensajes como Kafka / RabbitMQ, que luego fluyen a algún data-warehouse y finalmente son consultados por alguna UI de análisis. ¿Realmente deberíamos gastar el 50% de nuestro presupuesto para test en escribir test unitarios para una aplicación que esté centrada en la integración y apenas tenga lógica? A medida que aumenta la diversidad de tipos de aplicaciones (bots, criptografía, Alexa-skills), aumentan las posibilidades de encontrar escenarios en los que la pirámide de test no sea la mejor opción

Es hora de enriquecer el abanico de test y familiarizarse con más tipos de test (las siguientes secciones sugieren algunas ideas), modelos como la pirámide de test, pero también hacer coincidir los tipos de test con los problemas del mundo real al que te enfrentas ('Hola, nuestra API está rota, ¡escribamos contract testing dirigidos al consumidor!'), diversifica tus test como un inversor que construye una cartera de inversión basada en el análisis de riesgos — evalúa dónde pueden surgir problemas y combina algunas medidas de prevención para mitigar esos riesgos potenciales

Una advertencia: el TDD en el mundo del software adopta una cara de falsa dicotomía, algunos predicen que debemos usarlo en todas partes, otros piensan que es el diablo. Todos los que hablan en absoluto están equivocados:]

 **De lo contrario:** Te perderás algunas herramientas con un ROI increíble, algunas como Fuzz, lint y mutation pueden proporcionar valor en 10 minutos

►  Código de Ejemplo

○ 2.2 Los test de Componentes pueden ser tu mejor amigo

 Haz:

Cada test unitario cubre una pequeña parte de la aplicación y cubrirla totalmente cuesta muchísimo, mientras que los test end-to-end cubren fácilmente mucho terreno, pero son costosos y más lentos, ¿por qué no aplicar un enfoque equilibrado y escribir test más grandes que test unitarios pero más pequeños que los test end-to-end? Los test de componente es la canción no cantada del mundo del testing — proporcionan lo mejor de ambos mundos: rendimiento razonable y la posibilidad de aplicar patrones TDD + cobertura realista

Los test de componente se centran en la 'unidad' de microservicios, funcionan contra la API, no mockean nada que pertenezca al microservicio en sí (por ejemplo, base de datos real, o al menos la versión en memoria de esa base de datos) pero hace stub de cualquier cosa que sea externa como llamadas a otros microservicios. Al hacerlo, probamos lo que desplegamos, nos acercamos a la aplicación de fuera a dentro y obtenemos una gran confianza en un período de tiempo razonable

 **De lo contrario:** Puedes pasar muchos días escribiendo test unitarios para descubrir que solo tiene un 20% de cobertura del sistema

►  Código de Ejemplo

○ 2.3 Asegúrate de que las nuevas versiones no rompan el API usando tests de contrato

 **Haz:** Pongamos que tu microservicio tiene múltiples consumidores, y tenemos en ejecución diferentes versiones del servicio por compatibilidad (para que todos estén contentos). Luego cambias un campo y "¡boom!", uno de los consumidores que necesita ese campo se cabrea. Este es el Catch-22 del mundo de la integración: es muy difícil para el lado del servidor considerar todas las expectativas de todos los consumidores. Por otro lado, los consumidores no pueden realizar ningún test porque el servidor controla las fechas de release. [Los contratos dirigidos por el consumidor y el framework PACT] (<https://docs.pact.io/>) nacieron para regularizar este proceso con un enfoque muy disruptivo: no es el servidor quien define los test de sí mismo, sino que son los consumidores quienes definen los test de ¡el servidor! PACT puede registrar las expectativas del consumidor y dejarlas en una ubicación compartida, "broker", para que el servidor pueda cogerlas y cumplir con las expectativas y ejecutar cada construcción utilizando la librería PACT para detectar contratos incumplidos — una expectativa de consumidor no cumplida. Al hacerlo, todos los desajustes de la API cliente-servidor se detectan muy pronto durante la construcción / CI y pueden ahorrarte mucha frustración

 **De lo contrario:** Las alternativas son test manuales agotadores o miedo al despliegue

►  Código de Ejemplo

○ 2.4 Testea tus middlewares aisladamente

 **Haz:** Muchos evitan los test de middleware porque representan una pequeña porción del sistema y requieren ejecutar un servidor Express. Ambas razones son incorrectas — los middlewares son pequeños pero afectan a todas o la mayoría de las solicitudes y pueden testearse fácilmente como funciones puras que obtienen {req, res} objetos JS. Para testear una función de middleware se debe invocar y usar spy ([usando Sinon, por ejemplo] (<https://www.npmjs.com/package/sinon>)) sobre la interacción con los objetos {req, res} para garantizar que nuestra función middleware realiza la acción correcta. La librería [node-mock-http] (<https://www.npmjs.com/package/node-mock-http>) lo lleva aún más lejos y factoriza los objetos {req, res} ademas de añadir el spy. Por ejemplo, puede assertar si el estado http que se estableció en el objeto res coincide con el esperado (consulta el ejemplo a continuación)

 **De lo contrario:** Un bug en un middleware de Express === un bug todas o casi todas las peticiones

►  Código de Ejemplo

○ 2.5 Mide y refactoriza utilizando herramientas de análisis estático

 **Haz:** El uso de herramientas de análisis estático ayuda proporcionando formas objetivas para mejorar la calidad del código y a tener el código mantenible. Puede agregar herramientas de análisis estático a su pipeline de CI para abortar cuando encuentre code smells. Sus principales beneficios sobre el linter plano son la habilidad de analizar la calidad en el contexto de múltiples ficheros (por ejemplo encontrar duplicados), realizando análisis avanzados (por ejemplo complejidad del código) y siguiendo el historial y el progreso de cada problema. Dos ejemplos de herramientas que puedes usar son [Sonarqube](#) (2,600+ estrellas) y [Code Climate](#) (1,500+ estrellas)

Crédito: [Keith Holliday](#)

 **De lo contrario:** Con una mala calidad de código, los bugs y el rendimiento siempre serán un problema que ninguna librería completamente nueva o características punteras van a poder solucionar

►  Código de Ejemplo

○ 2.6 Comprueba tu predisposición al caos relacionado con Node

 **Haz:** Extrañamente, la mayoría de los test de software son acerca de lógica y datos, pero los errores más graves (y que son muy difíciles de resolver) son problemas de infraestructura. Por ejemplo, ¿alguna vez has testeado que pasa cuando se sobrecarga la memoria? ¿o cuando el servidor/proceso muere? ¿o tu sistema de monitorización es capaz de darse cuenta cuando la API es un 50% más lenta de lo normal? Para probar y evitar este tipo de problemas —[Chaos engineering](#) fue creado por Netflix. Su objetivo es proporcionar conciencia, frameworks, y herramientas para testear la resiliencia de nuestras aplicaciones en problemas caóticos. Por ejemplo, una de las herramientas más conocidas [the chaos monkey](#), mata servidores de forma aleatoria para comprobar si nuestro servicio aun puede dar servicio a los usuarios y asegurar que no depende de un solo servidor (hay también una versión para kubernetes, [kube-monkey](#), que mata pods en vez de servidores. Todas estas herramientas funcionan a nivel de hosting/plataforma, pero ¿qué pasa si deseas probar y generar caos a nivel Node puramente como comprobar como tu proceso Node hace frente a errores no controlados, o a rejects de promesas no capturados, o sobrecarga de la memoria de v8 por encima del máximo de 1.7GB o si la UX sigue siendo buena si se satura el event loop? Para todo esto he escrito [node-chaos](#) (alpha) que proporciona todo tipo de formas de crear el caos en Node

✖ De lo contrario: No hay escapatoria, la ley de Murphy afectará a producción sin piedad

▶  Código de Ejemplo

○ 2.7 Evitar fixtures globales y seeds, añade datos por cada test

 Haz: Siguiendo la regla de oro (sección 0), cada test debe añadir y actuar en su propio conjunto de filas en base de datos para evitar el acoplamiento y poder explicar fácilmente el flujo del test. En realidad muchos testers se saltan esta regla al añadir datos a la DB solo una vez antes de ejecutar los test ([también conocido como 'test fixture'](#)) a favor de mejorar el rendimiento. Si bien el rendimiento es una preocupación válida — puede mitigarse de otras formas (consulte la sección "test de componentes"), sin embargo, la complejidad del test es más dolorosa que otras consideraciones la mayoría de las veces. De manera práctica, haz que cada test añada explícitamente los registros que necesitas y actúe sobre ellos. Si el rendimiento se convierte en algo crítico — se puede llegar al compromiso de utilizar los mismos datos en un conjunto de test siempre que no se muten los datos (por ejemplo en queries)

✖ De lo contrario: Muchos test fallaran, un despliegue se aborta, nuestro equipo perderá mucho tiempo, ¿tenemos un bug? vamos a investigar, ah no — parece que dos test están mutando el mismo dato

▶  Código de Ejemplo

Sección 3 : Frontend Testing

○ 3.1 Separa la UI de la funcionalidad

 **Haz:** Al centrarnos en testear la lógica del componente, los detalles de la interfaz de usuario solo pueden entorpecernos, por lo que debes abstraerte de ellos y que los test se centren en datos puros. En la práctica, extrae los datos que necesites de una manera abstracta sin que este acoplada a la interfaz gráfica, haz aserciones de los datos puros (vs detalles visuales en HTML/CSS) y desactiva las animaciones que pueden hacer lenta la interfaz. En este punto podrías pensar en desactivar la interfaz y solo hacer test de la parte back del UI (por ejemplo servicios, acciones, store) pero esto solo dará como resultado test ficticios, diferentes a la realidad y no desvelaran casos en los que los datos correctos no llegan a la interfaz de usuario

 **De lo contrario:** Los datos calculados puros de tu test pueden estar listos en 10ms, pero luego todo el test tarda 500ms (100 test = 1 min) debido a alguna animación irrelevante

►  Código de Ejemplo

3.2 Consulta elementos HTML basándote en atributos que no deberían cambiar

 **Haz:** Consulta elementos HTML basándote en atributos que deberían permanecer intactos a cambios gráficos al contrario que selectores CSS y etiquetas de los formularios. Si el elemento designado no tiene esos atributos, crea un atributo dedicado solamente a los test como 'test-id-submit-button'. Seguir este patrón no solo asegura que tus test funcionales/lógica no se rompan nunca por cambios estéticos, sino que también queda claro a cualquier desarrollador que ese elemento y atributo están ahí por y para los test y no deben eliminarse

 **De lo contrario:** Quieres testear el login de tu app que tiene muchos componentes, lógica, servicios, y todo está bien configurado - stubs, spies, las llamadas Ajax están aisladas. Todo parece perfecto. Entonces el test falla porque el diseñador ha cambiado la clase de un div de 'thick-border' a 'thin-border'

►  Código de Ejemplo

3.3 Siempre que sea posible, testea con un componente real y totalmente renderizado

 **Haz:** Siempre que tenga un tamaño razonable, testea tu componente como lo hacen tus usuarios, renderiza completamente la interfaz de usuario, actúa sobre ella y comprueba que la interfaz se comporta como esperabas. Evita todo tipo de mocks, partials o shadow rendering - hacerlo puede provocar bugs no detectados debido a la falta de detalles y dificultará el mantenimiento a medida que los test interfieren con las partes internas (consulte la sección 'Acercarse al testing caja-negra'). Si uno de los componentes hijos ralentiza significativamente tu test (por ejemplo por una animación) o complica el setup, considera reemplazarlo explícitamente por un fake

Con todo esto también es necesario tener ciertas precauciones: esta técnica funciona para componentes pequeños / medianos que contienen un número razonable de componentes hijos. Renderizar completamente un componente con demasiados hijos hará que sea difícil analizar los fallos de los test (análisis de causa raíz) y puede ser demasiado lento. En estos casos, escribe los menos test que necesites contra ese componente principal y más test contra sus hijos

 **De lo contrario:** Al hurgar en las partes internas de un componente, invocando sus métodos privados y verificando el estado interno - tendrás que refactorizar todos los test siempre que refactorices los componentes ¿Realmente quieres dedicar ese tiempo en hacer este mantenimiento?

► Código de Ejemplo

3.4 No pauses, usa el soporte del framework para eventos asincronos e intenta acelerar las cosas

 **Haz:** En muchos casos, el tiempo que tarda una unidad bajo test es desconocido (por ejemplo, la animación elimina la visualización de un elemento) - en este caso, evita esperar (por ejemplo `stTimeOut`) y elige métodos mas deterministas que la mayoría de las plataformas proveen. Algunas librerías permiten esperar en operaciones (por ejemplo [Cypress cy.request\('url'\)](#)), otras proveen un API para esperar como [@testing-library/dom method wait\(expect\(element\)\)](#). A veces, una forma más elegante es hacer stub del recurso lento, como una API por ejemplo, con lo que el momento de respuesta se vuelve determinista, y volvemos a poder renderizar el componente directamente. Cuando tengas dependencias con algún componente externo que espera, puede ser útil modificar el tiempo con librerías como [hurry-up the clock](#). Esperar es algo que debemos evitar siempre, por que fuerza que tu test sea lento o tenga ciertos riesgos (cuando esperas un tiempo muy bajo). Siempre que sea inevitable esperar y hacer polling, y el framework de testing no nos da soporte, algunas librerías de npm pueden ayudar con soluciones semi-deterministas, como [wait-for-expect](#)

 **De lo contrario:** Cuando se espera mucho tiempo, los test serán un orden de magnitud más lentos. Cuando intentes esperar tiempos bajos, los test fallarán cuando la unidad bajo test no haya respondido a tiempo. Por tanto, se reduce a balancear entre puntos débiles y el rendimiento malo

►  [Código de Ejemplo](#)

3.5 Observa como se sirve tu contenido a nivel de red

 Example using Google LightHouse

 **Haz:** Usa un monitor que garantice que la carga de la página esté optimizada - esto incluye cualquier problema de UX como descarga lenta o un paquete no minimizado. El mercado de herramientas de este tipo no es pequeño: herramientas básicas como [pingdom](#), AWS CloudWatch, [gcp StackDriver](#) se puede configurar para ver si el servidor esta corriendo y respondiendo dentro de un SLA razonable. Esto tan solo ralla la superficie, podría haber muchísimas cosas mal, por tanto es mejor optar por herramientas especializadas en frontend, (por ejemplo [lighthouse](#), [pagespeed](#)) y que hagan un análisis mucho más amplio. El foco debe ponerse en los síntomas y métricas que afecten directamente a UX, como el tiempo de carga, [render mínimo](#), [tiempo hasta que la pagina es manejable \(TTI\)](#). Sobre todo esto, también debes estar atento a posibles causas técnicas, como garantizar que el contenido sea comprimido, tiempo hasta el primer byte, optimizar imágenes, asegurar un tamaño del DOM razonable, SSL, y muchos más. Es aconsejable tener toda esta monitorización durante el desarrollo, como parte del CI y mucho más importante - 24x7 en los servidores de producción / CDN

 **De lo contrario:** Es muy decepcionante darse cuenta de que después de haber tenido mucho cuidado y trabajo en crear una interfaz, pasen el 100% de los test funcionales y un pipeline sofisticado - el UX es horrible y lento por culpa de un CDN mal configurado

►  Código de Ejemplo

○ 3.6 Usa stubs para recursos lentos como el API de back-end

 **Haz:** Cuando programas tus test principales (no los test E2E), evita interactuar con cualquier recurso que este fuera de tu responsabilidad y control, como las API de back-end y usar stubs en su lugar (es decir, un doble). De forma práctica, en vez de hacer llamadas de red reales al API, utiliza alguna librería (como [Sinon](#), [Test dobles](#), etc) para stubear las repuestas de API. El principal beneficio es evitar la inestabilidad - testeando o suplantando APIs, por definición, no son muy estables y de vez en cuando fallarán los test, aunque TU componente se comporte bien (en producción generalmente se aceleran las respuestas, pero no esta pensado para hacer test). Hacerlo te permitirá simular varios comportamientos de API que deberían definir el comportamiento de nuestro componente para caminos no felices, por ejemplo cuando no se encuentran datos o cuando API devuelve un error. Por último, pero no por ello menos importante, las peticiones de red hacen más lentos nuestros test

 **De lo contrario:** La media de ejecución de los test no dura más de unos pocos ms, una llamada a API estándar dura 100ms>, lo que lo hace cada test ~20x más lento

►  Código de Ejemplo

3.7 Haz muy pocos test end-to-end que abarquen todo el sistema

 **Haz:** Aunque E2E (end-to-end) para algunos significa solo hacer test de UI en navegador de verdad (ver el punto 3.6), para otros significa que impliquen todo el sistema, incluido el backend real. Esto es muy valioso ya que te cubren errores de integración entre el frontend y el backend que pueden ocurrir por diferencias de opinión en el esquema de datos. También son un método eficiente para sacar errores de integración entre backends (por ejemplo, microservicio A envía datos erróneos al microservicio B) e incluso para detectar fallos de despliegue - actualmente no hay herramientas para test E2E solo backend tan amigables y maduras como las de UI como [Cypress](#) y [Puppeteer](#). La desventaja de estos test es su alto coste, tener un entorno configurado con todos los componentes, la fragilidad de los test - si tenemos 50 microservicios, solo con que falle uno, los test E2E fallan. Por estas razones tenemos que usar moderadamente esta técnica y probablemente tener entre 1 y 10 test de este tipo. Dicho esto, incluso una cantidad pequeña de test E2E es probablemente que detecten el tipo de problemas a los que están realmente dirigidos: despliegue e integración. Es aconsejable que se ejecuten en un entorno lo más parecido a producción

 **De lo contrario:** Puedes invertir mucho en testear la funcionalidad de la UI para darte cuenta demasiado tarde que el backend devuelve un contrato (el esquema de datos con el que la UI trabaja) muy diferente al esperado

3.8 Acelera los test E2E reutilizando las credenciales de login

 **Haz:** En los test E2E que involucren un backend real que usa un token para identificarse en las llamadas a API, no vale la pena aislar el test tanto como para que se cree un usuario y se haga login en cada test. En vez de esto, haz login una vez antes de ejecutar todos los test (en el before-all) guarda el token de forma local y reutilizalo en cada petición. Esto parece violar unos de los principios básicos - mantén los test autónomos sin acoplamiento de recursos. Y es cierto, pero en los test E2E el rendimiento es clave, y crear 1-3 peticiones a API antes de empezar cada test individual puede llevarnos a unos tiempos de ejecución horribles. Reutilizar las credenciales no significa que los test tengan que actuar sobre los mismos registros de usuario - si se basan en ellos (por ejemplo, testeando el historial de pagos), asegurate de crear los registro como parte del test y evita compartirlos con otros test. Y siempre recuerda que el backend puede ser sustituido - si tus test están focalizados en el frontend puede ser mejor aislarlos y desconectar las API de backend (consulta el punto 3.6)

 **De lo contrario:** Dados 200 test y asumiendo que un login son 100ms = 20 segundos solo para hacer el mismo login una y otra vez

►  Código de Ejemplo

○ 3.9 Haz un test E2E que navegue toda la página (smoke test)

 **Haz:** Para el monitoreo de producción y verificar que nada se rompe en tiempo de desarrollo (sanity check), ejecuta un único test E2E que visite todas o la mayoría de las páginas y se asegure que ninguna se rompe. Este tipo de test proporciona un gran retorno de la inversión ya que es un bastante sencillo de crear y mantener y puede detecta cualquier tipo de fallo, incluido funcionales, red y despliegue. Otras formas de hacer smoke y sanity checks no son tan confiables y exhaustivas - algunos equipos de operaciones simplemente hacen ping a la página de inicio (en producción) o desarrolladores que tiene muchos test de integración que no levanta errores de construcción o de navegador. No hace falta decir que este test no sustituye los test funcionales, solo sirven como detector de humo rápido

 **De lo contrario:** Todo puede parecer estar bien, todos los test pasan, el health-check de producción está ok también, pero el componente de pago se construyó mal y simplemente la ruta /Payment no se renderiza

►  Código de Ejemplo

○ 3.10 Exponer los test como un documento colaborativo vivo

 **Haz:** Además de aumentar la confiabilidad de la aplicación, los test te dan otra característica muy atractiva - sirven de documentación viva. Dado que los test hablan en un lenguaje menos técnico y sobre el producto y UX, usar las herramientas correctas puede servir como un artefacto de comunicación que alinea en gran medida a desarrolladores y su cliente. Por ejemplo, algunos frameworks permiten expresar el flujo y las expectativas (el test plan) utilizando un lenguaje legible para que cualquier stakeholder, incluyendo los product managers, pueden leer, aprobar y colaborar en los test convirtiéndose en el documento de requerimientos vivo. Esta técnica también se la conoce como 'test de aceptación', ya que permite al cliente definir sus criterios de aceptación en un lenguaje sencillo. Esto es [BDD \(behavior-driven testing\)](#) en su forma más pura. Uno de los frameworks más populares para esto es [Cucumber que tiene su sabor en JavaScript](#), ver el ejemplo más abajo. Otra forma similar pero diferente, [StoryBook](#), permite exponer los componentes UI como un catálogo gráfico, donde cualquiera puede recorrer los diferentes estados de cada componente (por ejemplo renderizar una cuadricula sin filtro, con múltiples filas o con ninguna, etc), ver como queda y como se activa ese estado - esto también puede atraer a la gente de producto pero sobre todo sirve como documentación viva para los desarrolladores que consumen esos componentes

 **De lo contrario:** Después de invertir los mejores recursos en los test, es una pena no aprovechar ese tiempo y ganar un gran valor como es la documentación

►  [Código de Ejemplo](#)

○ 3.11 Detecta problemas visuales con herramientas automatizadas

 **Haz:** Configure herramientas automatizadas para capturar screenshots de UI cuando se presenten cambios y detecte problemas visuales como contenido superpuesto o roto. Esto garantiza que no solo se muestren los datos correctos si no que el usuario los vea correctamente. Esta técnica no es ampliamente usada, nuestra mentalidad nos lleva a los test funcionales, pero es lo visual lo que el usuario experimenta y con la cantidad de dispositivos es relativamente fácil pasar por alto algunos bugs en la UI. Algunas herramientas gratuitas pueden proporcionar lo básico - generar y guardar screenshots para la inspección manual por una persona. Mientras este enfoque podría ser suficiente para aplicaciones pequeñas, no es valido como cualquier otro test manual que exige trabajo de una persona cada vez que algo cambia. Por otro lado, es bastante difícil detectar problemas de UI automáticamente debido a que no está claramente definido - aquí es donde interviene el campo de la 'Regresión Visual' a resolver este rompecabezas de comparar la UI antigua con los últimos cambios y detectar diferencias. Algunas herramientas OSS/gratuitas pueden proporcionar parte de esta funcionalidad (por ejemplo [wraith](#), PhantomCSS pero podrían conllevar un tiempo de configuración muy alto. Algunas herramientas comerciales (por ejemplo [Applitools](#), [Percy.io](#)) dan un paso más reducir la instalación y contener funciones avanzadas como interfaces de administración, alertas, captura inteligente que elimina el 'ruido visual' (por ejemplo, banners, animaciones) e incluso llegan a adelantar el análisis de la causa raíz de los cambios del DOM / css que han causado el problema

 **De lo contrario:** ¿Como de bien está hecha una pagina que muestra buen contenido (100% test ok), carga de forma instantánea pero la mitad del área de contenido está oculto?

- ▶  Código de Ejemplo

Sección : Midiendo la efectividad de los Test

4.1 Completa una cobertura suficiente que de confianza, ~80% parece el numero de la suerte

 **Haz:** El propósito de los test es tener suficiente confianza para moverse rápido, obviamente cuando más código se pruebe, más confianza tendremos en el equipo. La cobertura nos mide cuantas líneas de código (y ramas, declaraciones, etc) se alcanzan mediante los test. Entonces, ¿cuanta cobertura es suficiente? Obviamente 10-30% es demasiado bajo para tener alguna idea de que puedes tener que corregir, y por otro lado el 100% es muy caro y puede cambiar el foco de los caminos críticos a rincones apenas usados del código. La respuesta larga es que depende de muchos factores como el tipo de aplicación - si estas construyendo la siguiente generación del Airbus A380 un 100% es obligatorio, pero para una web de dibujos animados, el 50% podría hasta ser demasiado. Aunque la mayoría de los entusiastas de los test dicen que el porcentaje de cobertura correcto es contextual, la mayoría de ellos comentan que el 80% como la regla correcta ([Fowler: "in the upper 80s or 90s"](#)) que posiblemente debería satisfacer la mayoría de aplicaciones

Consejos de implementación: es posible que quieras configurar la integración continua (CI) para que tenga un umbral de cobertura ([Jest link](#)) y que pare el pipeline cuando no cumpla el estándar (también es posible configurar el porcentaje por componente, véase el ejemplo a continuación). Ademas de esto, deberías considerar detectar la bajada de cobertura (cuando un nuevo commit tiene menos cobertura que antes) - esto empujara a los desarrolladores a aumentar o al menos preservar la cantidad de código con test. Aunque también puede ser trampeado como se muestra en los siguientes puntos

 **De lo contrario:** La confianza y los números van de la mano, sin saber realmente que se ha testeado la mayor parte del sistema - habrá algo de miedo y el miedo te retrasará

►  Código de Ejemplo

○ 4.2 Inspecciona los reportes de cobertura para detectar áreas no testadas y otras cosas raras

 **Haz:** Algunos problemas se ocultan por debajo del radar y son realmente difíciles de encontrar utilizando herramientas tradicionales. Estos no son realmente bugs sino más bien comportamientos curiosos de la aplicación que podrían tener un gran impacto. Por ejemplo, a menudo algunas áreas de código no se invocan nunca o rara vez - puedes pensar que la clase 'PricingCalculator' siempre determina el precio del producto, pero resulta que en realidad nunca se invoca, aunque tenemos 10000 productos en base de datos y muchas ventas... Los reportes nos ayudan a darnos cuenta de si la aplicación se comporta de la manera que esperamos. Aparte de eso, también podemos resaltar qué tipos de código no se testean: que el 80% del código se testea, no nos indica si las partes críticas están cubiertas. Generar reportes es fácil: simplemente ejecute su aplicación en producción o durante test con cobertura y luego revisa los reportes que resaltan la frecuencia con la que se invoca cada parte del código. Si le dedicas un tiempo para echar un vistazo a estos datos, puedes encontrar algunas errores

 **De lo contrario:** Si no sabes qué trozos de código no se testean, no sabes dónde pueden aparecer problemas

►  Código de Ejemplo

4.3 Mide la cobertura lógica usando mutation testing

 **Haz:** La métrica de cobertura tradicional a menudo miente: puede mostrarle una cobertura de código del 100%, pero ninguna de sus funciones, ni siquiera una, devuelve la respuesta correcta. ¿Cómo? simplemente mide sobre qué líneas de código se pasó en los test, pero no verifica si los test realmente han comprobado algo - asumiendo la respuesta correcta. Como alguien que viaja por negocios y muestra su pasaporte, esto no te asegura que haya realizado ningún trabajo, solo que ha visitado ciertos aeropuertos y hoteles

Los test basados en mutaciones nos ayudan midiendo la cantidad de código que en realidad se TESTEÓ, no solo VISITADO. [Stryker] (<https://stryker-mutator.io/>) es una librería JavaScript para mutation testing y la implementación es realmente clara:

(1) cambia intencionalmente el código y "planta bugs". Por ejemplo, el código `newOrder.price === 0` se convierte en `newOrder.price! = 0`. Estos "bugs" se llaman mutaciones

(2) ejecuta los test, si todo va bien, entonces tenemos un problema - los test no cumplen su propósito de descubrir bugs, las mutaciones se denominan supervivientes. Si los test fallaron, entonces genial, las mutaciones fueron destruidas

Saber que todas o la mayoría de las mutaciones fueron destruidas da mucha más confianza que la cobertura tradicional y el tiempo de configuración es muy similar

 **De lo contrario:** Te engañas si crees que una cobertura del 85% significa que tus test detectarán errores en el 85% de tu código

►  Código de Ejemplo

4.4 Prevención de problemas de código de test con linters para test

 **Haz:** ESLint tiene un conjunto de plugins específicos para inspeccionar patrones de código de test y descubrir problemas. Por ejemplo, [eslint-plugin-mocha] (<https://www.npmjs.com/package/eslint-plugin-mocha>) avisará cuando un test se escriba a nivel global (no es hijo de un describe ()) o cuando se omiten los test (<https://mochajs.org/#inclusive-tests>), lo que puede llevar a creer erróneamente de que todos los test están ok. Del mismo modo, [eslint-plugin-jest] (<https://github.com/jest-community/eslint-plugin-jest>) puede, por ejemplo, advertir cuando un test no tiene aserciones (sin verificar nada)

 **De lo contrario:** Ver un 90% de cobertura de código y 100% de test verdes te provocara una sonrisa hasta que te das cuenta de que muchos test no asercionan nada y muchos test simplemente se omitieron. Con suerte, no desplegaste nada basándote en esta falsa observación

►  Código de Ejemplo

Sección 5 : CI y otras medidas de calidad

5.1 Enriquece tus linters y cancela las construcciones que tienen problemas de linter

 **Haz:** Los linters son comida gratis, con una configuración de 5 minutos, obtienes gratis un piloto automático que vigila tu código y detecta problemas importantes mientras escribes. Atrás quedaron los días en los que el linter era solo maquillaje (¡no hay punto y coma!). Hoy en día, los linters pueden detectar problemas graves como errores que no se lanzan correctamente y perder información. Además de su conjunto básico de reglas (como [standar] (<https://www.npmjs.com/package/eslint-plugin-standard>) o [Airbnb] (<https://www.npmjs.com/package/eslint-config-airbnb>)), considera incluir algunos conjuntos de reglas especializadas como [eslint-plugin-chai-expect] (<https://www.npmjs.com/package/eslint-plugin-chai-expect>) que pueden descubrir test sin aserciones, [eslint-plugin-promise] (<https://www.npmjs.com/package/eslint-plugin-promise?activeTab=readme>) puede descubrir promesas sin resolución (tu código nunca continuará), [eslint-plugin-security] (<https://www.npmjs.com/package/eslint-plugin-security?activeTab=readme>) que puede descubrir expresiones regulares que podrían usarse para ataques DOS, y [eslint-plugin-you-dont-need-lodashunderscore] (<https://www.npmjs.com/package/eslint-plugin-you-dont-need-lodashunderscore>) es capaz de avisar cuando el código utiliza métodos de librerías de utilidades que forman parte de V8, métodos básicos como Lodash._map (...)

 **De lo contrario:** Considera un día lluvioso donde producción sigue fallando pero los registros no muestran el call stack de errores. ¿Qué pasa? Tu código emite por error un objeto sin error y se perdió el trazado, una buena razón para golpearse la cabeza contra una pared. Una configuración de linter de 5 minutos podría detectar este GAZAPO y salvarte el día

- ▶  Código de Ejemplo

5.2 Acorta el tiempo de feedback con local developer-CI

 Haz: ¿Tienes un pipeline de CI con test, linter, verificación de vulnerabilidades, etc?

Ayuda a los desarrolladores a ejecutarlo también localmente para solicitar comentarios instantáneos y acortar el [ciclo de feedback] (<https://www.gocd.org/2016/03/15/are-you-ready-for-continuous-delivery-part-2> -circuitos de retroalimentacion/). ¿Por qué? un proceso de testing eficiente constituye muchos bucles iterativos: (1) test -> (2) feedback -> (3) refactor. Cuanto más rápido sea el feedback, más iteraciones de mejora puede realizar un desarrollador por módulo y perfeccionar los resultados. Por otro lado, cuando el feedback tarda en llegar, se podrían realizar menos iteraciones de mejora en un solo día, el equipo podría estar ya haciendo otra cosa / tarea / módulo y podría no estar listo para refinar ese módulo

En la practica, algunos proveedores de CI (Ejemplo: [CircleCI CLI local] (<https://circleci.com/docs/2.0/local-cli/>)) permiten ejecutar el pipeline localmente. Algunas herramientas comerciales como [wallaby proporcionan información valiosa y de test] (<https://wallabyjs.com/>) para el desarrollador sin coste. Alternativamente, puedes agregar scripts npm en el package.json para ejecutar todos los comandos de calidad (por ejemplo, test, linter, vulnerabilidades) - usa herramientas como [concurrently] (<https://www.npmjs.com/package/concurrently>) para paralelizarlas y que el código de salida sea distinto de cero si falla alguna de las herramientas. Ahora el desarrollador solo debe invocar un comando - por ejemplo "npm run quality": para obtener feedback en el acto. Considera también cancelar un commit si el control de calidad falla usando un githook ([husky puede ayudar] (<https://github.com/typicode/husky>))

 De lo contrario: Cuando los resultados de calidad llegan un día más tarde que el código, los test no se convierten en una parte fluida del desarrollo, sino en algo formal posterior al mismo

►  Código de Ejemplo

○ 5.3 Realiza test e2e sobre un espejo de producción

 **Haz:** Los test End to end (e2e) testing son el principal desafío de un pipeline CI - crear un entorno que sea un espejo de producción, efímero, que se genere sobre la marcha con todos los servicios necesarios puede ser tedioso y muy costoso. Buscando la mejor opción: [Docker-compose](#) permite crear un entorno dockerizado aislado con mismos contenedores usando un único fichero de texto plano, pero con la tecnología por debajo (redes, despliegues) ser diferentes a las de producción. Puedes combinarlo con '[AWS Local](#)' para trabajar en hacer stubs de servicios AWS de verdad. Si usas [serverless](#) existen múltiples frameworks como serverless y [AWS SAM](#) que te permiten invocar código Faas en local

El enorme ecosistema de Kubernetes aún no tiene una herramienta estándar para la duplicación en local y CI, aunque salen herramientas nuevas cada día. Un enfoque puede ser ejecutar implementaciones de 'kubernetes reducidos' con [Minikube](#) o [MicroK8s](#) que es igual que el kubernetes completo pero con menos complejidad. Otro enfoque es tener un kubernetes de verdad remoto para test, algunos proveedores de CI (por ejemplo [Codefresh](#)) tienen integración nativa con entornos kubernetes y pueden fácilmente ejecutar el pipeline CI sobre algo más real, otros permiten ejecutar scripts personalizados contra kubernetes remotos

 **De lo contrario:** El uso de diferentes tecnologías para la producción y los test exige mantener dos modelos de implementación y mantiene a los desarrolladores y al equipo de operaciones separados

►  Código de Ejemplo

5.4 Paralelizar la ejecución de los test

 **Haz:** Cuando se hace correctamente, los test son tu amigo 24/7 proporcionando feedback instantáneo. En la práctica, ejecutar 500 test unitarios en un solo proceso en CPU puede llevar demasiado tiempo. Afortunadamente, los test runner más modernos y las plataformas de CI (como [Jest](#), [AVA](#) y [Mocha extensions](#)) pueden paralelizar los test en múltiples procesos y lograr una mejora importante en el tiempo en entregar feedback. Algunos proveedores de CI también paralelizan los test en contenedores (!) lo que acorta aún más la entrega de feedback. Ya sea localmente con múltiples procesos, o sobre algún CLI en cloud usando múltiples máquinas - paraleliza manteniendo los test autónomos para que cada uno pueda ejecutarse en diferentes procesos

 **De lo contrario:** Obtener el resultado de los test en 1 hora después de hacer push de código nuevo, mientras desarrollas nuevas funcionalidades, es la mejor receta para quitarle relevancia a los test

►  Código de Ejemplo

5.5 Mantente al margen de problemas legales usando la verificación de licencia y plagio

 **Haz:** Los problemas de licencias y plagio seguramente no son tu prioridad ahora mismo, pero ¿por qué no hacer check en esta tarea en solo 10 minutos? Existen muchos paquetes npm como [license check](#) y [plagiarism check](#) (commercial pero con plan gratuito) que puedes integrar fácilmente en tu pipeline CI e inspeccionar en busca de problemas como dependencias con licencias restrictivas, o código que fue copiado y pegado de stackoverflow que aparentemente viola algunos copyrights

 **De lo contrario:** Sin querer, los desarrolladores pueden usar paquetes con licencias inapropiadas o copiar y pegar código comercial y encontrarse con problemas legales

►  Código de Ejemplo

5.6 Verifica constantemente las dependencias vulnerables

 **Haz:** Incluso las dependencias más conocidas como express, tiene vulnerabilidades conocidas. Puedes dominarlas fácilmente usando herramientas libres como [npm audit](#), o comerciales como [snyk](#) (que tiene una versión community gratuita). Ambas pueden ser ejecutadas desde tu CI en cada construcción

 **De lo contrario:** Mantener tu código limpio de vulnerabilidades sin herramientas dedicadas requiere que estés revisando constantemente las nuevas versiones. Muy tedioso

►  Código de Ejemplo



5.7 Automatiza la actualización de dependencias

 **Haz:** La introducción en yarn y del package-lock.json de npm introduce un desafío muy importante (el camino al infierno esta lleno de buenas intenciones)—por defecto ahora los paquetes no se auto actualizan más. Incluso en un equipo que ejecute un 'npm install' limpio y 'npm update' no van a caer nuevas actualizaciones. Esto conduce a versiones de paquetes desactualizadas en el mejor de los casos, y en el peor, a código vulnerable. Los equipos pasan a depender de la buena voluntad y memoria de los desarrolladores para actualizar el package.json a mano o a utilizar herramientas como [ncu](#) manualmente. Una formula mucho mejor seria automatizar el proceso de actualizar las versiones de las dependencias en las que más confiamos, pero no hay una solución perfecta, existen dos caminos para esta actualización:

(1) Podemos hacer que el CI falle con dependencias obsoletas—usando herramientas como '[npm outdated](#)' o '[npm-check-updates \(ncu\)](#)'. Hacerlo obligará a los desarrolladores a actualizar las dependencias

(2) Usar alguna de las herramientas comerciales que escanean nuestro código y envían automáticamente pull-request con actualización de dependencias. Una pregunta interesante que nos queda es cual va a ser la política de estas actualizaciones: si actualizamos cada parche se genera mucha sobrecarga, y hacerlo cuando haya una versión mayor nos lleva directos a usar versiones inestables o incompatibles (mucho paquetes muestran vulnerabilidades justo después de salir una versión nueva [lee sobre el incidente de eslint-scope](#))

Una política de actualizaciones eficiente puede permitir cierto 'periodo de concesión' - deja pasar versiones quedándote por detrás de @latest un tiempo antes de considerar que tu versión en local está obsoleta (por ejemplo, la versión que tienes en local es 1.3.1 y la versión del repositorio del paquete es 1.3.8)



De lo contrario: Producción estará ejecutando versiones de paquetes que han sido marcadas por los propios autores como versiones con riesgos

►  Código de Ejemplo



5.8 Otros consejos de CI no relacionados con Node

 **Haz:** Esta publicación se centra en los consejos sobre testing que están relacionados, o al menos pueden ejemplificarse con Node JS. Sin embargo, este punto agrupa algunos consejos no relacionados con Node que son bien conocidos

1. Usa una sintaxis declarativa. Esta es la única opción para la mayoría de los pipelines de CI, pero las versiones antiguas de Jenkins permiten usar código o incluso una interfaz de usuario
2. Opta por un CI que tenga soporte nativo de Docker
3. Falla antes, ejecuta tus test más rápidos primero. Crea un paso / hito de 'Smoke testing' que agrupe múltiples verificaciones rápidas (por ejemplo, linter, test unitarios) y que proporcione comentarios rápidamente al desarrollador que haya commiteado
4. Facilita la exploración de todas las partes de construcción, incluidos reportes de test, informes de cobertura, informes de mutación, registros, etc.
5. Crea múltiples pipelines / jobs para cada evento y reutiliza los pasos entre ellos. Por ejemplo, configura un job para confirmaciones de commits a ramas de desarrollo y uno diferente para la rama master. Permite la reutilización de la lógica usando pasos compartidos en los pipelines (la mayoría de los proveedores proporcionan algún mecanismo para la reutilización de código)
6. Nunca uses secretos directamente en la declaración del job, traelos de un store de secretos o de la configuración del propio job
7. Suba explícitamente la versión de una release o al menos asegúrate de que el desarrollador lo hizo
8. Construye solo una vez y realiza todas las inspecciones sobre el artefacto construido único (por ejemplo, imagen Docker)
9. Testea en un entorno efímero que no arrastra el estado entre las construcciones. El almacenamiento en caché de node_modules podría ser la única excepción

 **De lo contrario:** Echarás de menos años de sabiduría

5.9 Build matrix: Ejecuta los mismos pasos de CI usando múltiples versiones de Node

 **Haz:** En control de calidad influye totalmente la casualidad, cuanto más terreno cubras, más suerte tendrás al detectar problemas temprano. Al desarrollar paquetes reutilizables o ejecutar un producto multicliente con varias configuraciones y versiones de Node, el CI debe ejecutar el pipeline de test sobre todas las combinaciones de configuraciones. Por ejemplo, suponiendo que usemos MySQL para algunos clientes y Postgres para otros, algunos proveedores de CI admiten una característica llamada 'Matrix' que permite ejecutar todos los test contra todas las combinaciones de MySQL, Postgres y versiones de Node múltiples como 8, 9 y 10. Esto se hace utilizando solo configuración, sin ningún esfuerzo adicional (suponiendo que tengas test o cualquier otro control de calidad). Otros CI que no admiten Matrix pueden tener extensiones o ajustes para permitirlo

 **De lo contrario:** Entonces, después de hacer todo ese arduo trabajo de escribir los test, ¿vamos a dejar que los errores se cuelen solo por problemas de configuración?

►  Código de Ejemplo

Equipo

Yoni Goldberg



Papel: Escritor

Acerda de: Soy un consultor independiente que trabaja con empresas de fortune 500 y startups en garajes para pulir sus aplicaciones JS & Node.js. Más que ningún otro tema me fascina y tengo como objetivo dominar el arte del testing. También soy el autor de [Node.js Buenas Practicas](#)

 **Curso Online:** ¿Te gustó esta guía y deseas llevar tus habilidades de testing al máximo? Considera visitar mi curso completo [Testing Node.js & JavaScript From A To Z](#)

Sigeme:

-  [Twitter](#)
-  [Contact](#)
-  [Newsletter](#)

Bruno Scheufler

Rol: Revisor técnico y asesor

Se encargó de revisar, mejorar, lintear y pulir todos los textos

Acerda de: Ingeniero full-stack web, entusiasta de Node.js y GraphQL

Ido Richter

Rol: Concepto, diseño y buenos consejos

Acerda de: Un desarrollador frontend inteligente, experto en CSS y friki de los emojis

Kyle Martin

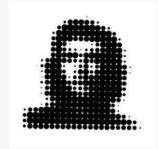
Rol: Ayuda a mantener este proyecto en funcionamiento y revisa los test relacionadas con la seguridad

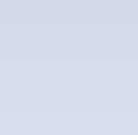
Acerda de: Le encanta trabajar en proyectos Node.js y seguridad de aplicaciones web

Contribuyentes



¡Gracias a estas personas maravillosas que han contribuido a este repositorio!

 Scott Davis 	 Adrien REDON 	 Stefano Magni 	 Yeoh Joer 	 Jhonny Moreira 	 Ian Germann 	 Hafez 
 Ruxandra Fediuc 	 Jack 	 Peter Carrero 	 Huhgawz 	 Haakon Borch 	 Jaime Mendoza 	 Cameron Dunford 

						
John Gee 	Aurelijus Rožėnas 	Aaron 	Tom Nagle 	Yves yao 	Userbit 	Glaucia 
						
kooge 	Michal 	roywalker 	dangen 	biesiadamich 	Yanlin Jiang 	sanguino 
						
Morgan 	Lukas Bischof 	JuanMa Ruiz 	Luís Ângelo Rodrigues Jr. 	José Fernández 	Alejandro Gutierrez Barcenilla 	Jason 