**TEST UNITARIO**

[**https://diegolaballos.com/podcast/40-que-es-y-para-que-sirve-el-testing-en-programacion/**](https://diegolaballos.com/podcast/40-que-es-y-para-que-sirve-el-testing-en-programacion/)

**¿QUE ES TESTEO?**

Las pruebas de aplicaciones web permiten a las empresas simular picos en el tráfico del usuario, explorar las vulnerabilidades de seguridad en un servidor de aplicaciones y probar la funcionalidad de las aplicaciones. A medida que más aplicaciones se migran a la nube, las pruebas de aplicaciones web pueden ayudar a los equipos de TI de las empresas a abordar los problemas de rendimiento, fiabilidad y seguridad que, inevitablemente, pueden surgir.

Habitualmente se tiende a definir testing/testeo (de aquí en más tratados como sinónimos) como “probará en ejecución el programa para ver si funciona correctamente”, o “es un proceso que lleva a confiar que el programa hará lo que se supone que debe hacer”. Pero esto es un grave error, ya que realizar testing implica exactamente lo contrario, testear es el “proceso de ejecutar un programa con el fin de encontrar errores”

Como su propio nombre indica, es el proceso de testear o de probar que algo funciona como debería.

Por ejemplo, imaginemos un fabricante de coches que está instalando un sistema de airbag. Evidentemente, este sistema se tendrá que probar antes de vender el coche, tendrán que hacer una serie de pruebas para demostrar y asegurarse que ese airbag funciona.

**¿QUE ES TESTEO EN PROGRAMACION?**

El testing en programación se refiere a la comprobación de que todo el código que se ha escrito funciona.

Por ejemplo, si estamos diseñando una app para pedir taxis, tendremos que comprobar que cuando el usuario abre la app, le aparece un mapa con su posición actual y un botón para pedir un taxi; que al apretar el botón de “pedir un taxi” lo lleva a una pantalla para elegir la hora; que después va a la pantalla de confirmar y que, cuando acaba el viaje, le sale el botón de pagar.

Pero ojo, también probamos si la app responde bien ante los errores. Por ejemplo, que, si no hay taxis, le aparezca el mensaje de error correspondiente; que si no tiene internet le aparezca un mensaje indicándolo; que cuando el pago falla, le aparezca una interfaz para volverlo a intentar.

Se trata de comprobar que, ante todas las situaciones a las que se puede enfrentar vuestra app, la respuesta es la correcta en cada caso.

**UNIT TESTING (TESTEO UNTARIO)**

Podríamos decir, que es el proceso de escribir código que compruebe el código original del proyecto. Sí, sé que esto puede ser difícil de entender, pero, es así.

Podríamos pensar en una aplicación como si fuera una casa: si una casa está compuesta de muchos pequeños ladrillos, una aplicación/software está compuesta de muchos pequeños trozos de código que, juntos, forman lo que vemos en la app.

El unit testing, lo que testea es ver si cada “ladrillo” está bien puesto/nivelado. Lo que haremos, será escribir código extra que se encarga de comprobar que cada ladrillo que ponemos está bien. De esta manera, si alguien coloca un ladrillo mal o lo modifica en algún momento, nos centraremos en ejecutar rápidamente estos tests automáticos.

Por ejemplo, imaginemos que cuando pedimos un taxi, se te da a elegir entre varios taxistas y te los lista por distancia en kilómetros.

Seguramente, en el código de la aplicación, habrá una pequeña función que se encargará de transformar dos posiciones GPS (que vendrían en un formato de latitud y longitud) a la distancia entre ellas. Por ejemplo, si el conductor está en la latitud X y longitud Y; y, el usuario en la latitud Z y longitud R, la distancia entre ellos, será M.

Si tenemos esta función dentro de nuestro código y queremos hacer un unit testing, lo que haríamos sería escribir un código que diría: si meto esta latitud y esta longitud, me tendrías que devolver esta distancia en kilómetros; si meto esta otra latitud y esta otra longitud, me tendrías que dar esta otra distancia en kilómetros; y, si meto una latitud y longitud inválidas, me tendrías que devolver un error.

Una vez que tengamos esto, podemos ejecutar este código para comprobar que todo es cierto y ver que todo está bien.

Otro ejemplo: cada vez que hagamos un cambio en el código, podemos volver a ejecutar todos nuestros tests para ver que todo sirve como debería. Podemos imaginarlo como una calculadora: supongamos que haces una operación complicada en la cabeza y luego vas a la calculadora para comprobar que está bien.

Esto, es el unit testing a grandes rasgos. Quizás ahora no se entiende al 100% la razón por la cual se hace, porque puedes decir: ¿para qué vas a comprobar algo que acabas de hacer y que se supone que funciona? Pero calma. En la última parte, veremos todas las razones por las cuales, se hace esto.

**AHORRO DE TIEMPO**

Sí, es un poco contradictorio con lo que acabo de decir, pero el testing puede hacerte ganar tiempo.

Hay que imaginar, especialmente, el unit testing y el UI testing como algo rápido; como algo que, al apretar un botón, obtienes cientos de resultados de tus tests. Son pruebas que, si tuvieras que hacer tú a mano, tardarías muchísimo más tiempo.

Imagina que estás haciendo una actualización de tu aplicación y empiezas a hacer varios cambios en tu app. Cuando empiezas a tocar código pueda pasar que, al cambiar algo, por accidente cambies otra cosa.

Pongamos un ejemplo del mundo real: imagina que quieres poner un cuadro en tu pared y, para ello, taladras, pones un tornillo y hecho. Tu cuadro está perfecto, pero, te das cuenta que te has emocionado tanto que has traspasado la pared de la otra habitación. Como sabes que eso es algo que puede pasar, cada vez que quieras colgar un cuadro, vas a ir a ver a la habitación contraria que todo esté bien.

Si ese ejemplo lo pasamos al código, es como pensar que se han colgado muchos cuadros en muchas habitaciones y hay que ir una por una a mirar que no la hayas liado. Aquí, te doy dos opciones: la primera, es que lo hagas tú mismo y vayas una por una a comprobar; la segunda, es tener un equipo de 20 personas para que, cada vez que cuelgues un cuadro, vayan cada uno a ver qué ha pasado.

**DETECTAR DÓNDE ESTÁN LOS ERRORES**

Va de la mano del punto anterior. Volvamos a los cuadros de la casa: imagina una casa de 1.000 habitaciones y que tu trabajo es colgar cuadros. El dueño de la casa te pide que cuelgues un cuadro en cada habitación. Una vez que terminas el trabajo, el dueño te dice que uno está mal. Tu le preguntas ¿Dónde?, y él te dice: Pues no lo sé. Hay muchas habitaciones y no recuerdo dónde era.

Tienes un problema y no sabes dónde está el error. Como comentábamos en el primer punto, vas a tardar tiempo extra en detectarlo y quizás termines por tocar un cuadro que estaba bien; o arreglar uno solo, cuando en realidad eran dos.

**ASEGURAR QUE LA APP FUNCIONA EN CUALQUIER MÓVIL**

Este beneficio, tiene mucho sentido en los dispositivos móviles. Para entenderlo, hay que saber cómo es el entorno móvil y que problema hay. En el caso de Android, particularmente, hay muchos fabricantes, muchos dispositivos y muchas versiones del sistema operativo.

Esto quiere decir, que hay múltiples combinaciones de todos ellos. Por ejemplo, Samsung S6 con Android 8; Nexus 5 con Android 7; Google Pixel con Android 6, etc. Pues bien, resulta que una función que funciona perfectamente en determinado móvil con determinado sistema puede no funcionar en otro móvil con otro sistema.

Es decir, que algo que funciona perfectamente en un Samsung S7, podría no funcionar en un Nexus 6; este es el problema. Puede ser por cualquier motivo, pero, el caso es que ciertas funciones pueden dar problemas en distintos entornos. Que algo funcione en cierto móvil, no significa que funcione en otro, por mucho que sea Android.

Dicho esto, tienes 3 opciones básicamente. La primera es probar tu aplicación en tu móvil, y listo; confiar en que funcionará en los demás y, si no lo hiciera, ya lo solucionarás. Evidentemente esta no es la mejor opción.

La segunda opción, sería probar la aplicación en tu móvil, y en tres o cuatro más. Tal vez un móvil viejo que tenías guardado; el de tu amigo, hermano, novia, padre, etc. De esta manera, reduces un poco el riesgo, pero sigues teniendo dos problemas: la pérdida de tiempo que te supondrá instalar y probar manualmente la aplicación en cada móvil; y otro, que es el hecho de que probar 3 o 4 móviles, no implica un gran progreso.

Aunque quisieras comprar más móviles para seguir con este método, per no sé hasta qué punto sería sostenible gastarte 10.000 euros en 150 móviles.

Como solución, el testing te permite automatizar todas estas pruebas y que lo haga una máquina por ti. Esto ya lo sabemos. Sin embargo, existen servicios que te permiten automatizar la ejecución de todos estos tests en móviles “remotos”.

**¿PARA QUÉ SIRVEN LOS TEST UNITARIOS?**

Los test unitarios sirven para asegurarnos de que un bloque de nuestro código, función o clase, funcionan correctamente y abarca la mayoría de casos de uso que se puedan dar. Además, proporciona robustez y calidad a nuestro código y confirma que funciona correctamente.

<https://cleformacion.com/tic-tek/-/blogs/testing-de-aplicaciones-web>

**TIPOS DE PRUEBAS**

**Pruebas de Aceptación.** Pruebas que realizamos con la finalidad de verificar si el producto cumple todos los requisitos especificados por el cliente, sus reglas de negocio. Debemos tratar la aplicación como una caja opaca en la que ingresamos valores y controlamos su respuesta.

**Pruebas de Sistemas.** En este tipo de pruebas evaluamos varios aspectos:

**Usabilidad:** se refiere a la capacidad de un software de ser comprendido, aprendido, usado y atractivo para el usuario, en condiciones específicas de uso. En este tipo de pruebas comprobamos el comportamiento del usuario dentro de la aplicación, si le resulta fácil de usar, si cuando empieza un flujo de trabajo no lo termina a medias, si no se pierde dentro de nuestra aplicación. En definitiva, si interactúa de manera fácil y sencilla con nuestra aplicación.

**Interfaz de usuario:** en estas pruebas evaluamos, por ejemplo, el tamaño de las ventanas, tipos de letras, colores, ubicación de los controles…

**Rendimiento:** se evalúa el tiempo de respuesta de nuestra aplicación web, de nuestros componentes y funcionalidad. Comprobamos el comportamiento del sistema ante diferentes situaciones.

**Configuración:** validamos que nuestra aplicación web funciona correctamente para distintos tipos de (en nuestro caso) navegadores web (firefox, chrome, internet explorer…)

**Pruebas de Integración.** Evaluamos cómo se integran cada una de las partes de nuestra aplicación, como interactúan entre ellas. Puede ser que varias partes por separado pasen todos nuestros tests, pero que a la hora de comunicarse entre sí no lo hagan bien, gracias a estas pruebas detectamos este tipo de errores.

**Pruebas de Regresión**. Son un tipo particular de pruebas. El objetivo es probar un error detectado en nuestra aplicación, corregirlo y volver a ejecutar la prueba para ver si la aplicación responde correctamente. Es importante en este tipo de pruebas realizar todos los tests ya hechos en nuestra aplicación, y comprobar que, al corregir el error, no nos hemos cargado nada por el camino.

<https://www.yeeply.com/blog/que-son-pruebas-unitarias/>

**BUENAS PRÁCTICAS PARA LOS TESTS UNITARIOS**

Una vez te acostumbres a realizar este tipo de pruebas irás descubriendo todas sus ventajas. Sin embargo, si todavía no tienes experiencia, vamos a ver algunos ejemplos de buenas prácticas en las pruebas unitarias de software.

Las pruebas unitarias deberían ser independientes. Si se produce cualquier tipo de mejora o cambio en los requerimientos, las pruebas unitarias no deberían verse afectados.

Prueba sólo un código a la vez.

Sigue un esquema claro. Puede parecer algo secundario, pero no lo es. Sé también consistente a la hora de nombrar tus unit tests.

Cualquier cambio necesita pasar el test. En el caso de producirse un cambio en el código de cualquier módulo, asegúrate de que hay una prueba unitaria que se corresponda con ese módulo y que este pasa las pruebas antes de cambiar la implementación.

Corrige los bugs identificados durante las pruebas antes de continuar. Asegúrate de realizar esta corrección antes de proseguir con la siguiente fase del ciclo de vida del desarrollo de software.

Acostúmbrate a realizar pruebas regularmente mientras programas. Cuanto más código escribas sin testar, más caminos tendrás que revisar para encontrar errores.

No puedes esperar que las pruebas unitarias descubran todos los errores de un software, pero sí que ahorran mucho tiempo al facilitar localizar los errores de una manera más sencilla.

**HERRAMIENTAS**

<https://www.paradigmadigital.com/dev/test-unitarios-javascript-introduccion/>

**¿QUÉ ES UN FRAMEWORK DE PRUEBAS UNITARIAS?**

Un framework de pruebas unitarias es una herramienta que nos permite escribir pruebas sobre un bloque de código, ejecutándolo bajo un entorno de javascript sin necesidad de interferir en el IDE ni en la propia aplicación.

**¿QUÉ DEBERÍA CUBRIR MI PRUEBA UNITARIA?**

Nuestra prueba unitaria deberá cubrir todas las posibles funcionalidades dentro de un bloque de código. Por ejemplo, dado el siguiente código, vamos a analizar qué casos de uso nos hacen falta:

**¿QUÉ FRAMEWORK DE PRUEBAS UNITARIAS DEBERÍA USAR?**

Actualmente, disponemos de muchos frameworks para realizar las pruebas de nuestro código en Javascript, estos son los más populares del año pasado:

* **Jest**
* Es el framework de test de Facebook, con el que se hacen los test de React, pero también se puede usar con otros frameworks como Vue o Angular, incluso con VanillaJs.
* Es muy rápido y muy fácil de aprender si empiezas con los test.
* Nos permite hacer test por snapshot (test para asegurarnos de que nuestra interfaz de usuario no cambia).
* Nos da la posibilidad de crear Spies para métodos y mocks de manera muy sencilla.
* **Mocha**
* Es uno de los frameworks más usados desde hace bastantes años.
* Se puede usar junto con otros frameworks de test, como Chai o Enzyme.
* Dispone de una gran cantidad de documentación.
* Es una herramienta muy flexible debido a la gran comunidad que lo respalda.
* **Jasmine**
* Es el framework recomendado por Angular para los test unitarios.
* Es fácil empezar con este framework, con una curva de aprendizaje suave.
* Jest está basado en Jasmine.

Aunque podríamos analizar uno a uno dichos frameworks, como el objetivo del post es aprender a escribir bien nuestros test, optaremos por Jest y veremos cómo escribir correctamente un test en dicho framework.

**CONCLUSIONES**

Escribir un test unitario no es algo complejo, pero sí que lleva su tiempo. Normalmente, si en realizar una funcionalidad se tarda 1 hora, para realizar los test de dicha funcionalidad se suele calcular multiplicando por tres el tiempo de desarrollo, ya que tenemos que abarcar, a ser posible, un mínimo del 90% de los posibles casos que se puedan dar: por ejemplo, cuando el código se ejecuta correctamente o cuando queremos controlar una excepción.

Todo ello conlleva unas acciones que, para tener un test correcto, hay que cubrir en la mayoría de las posibilidades o casos que se puedan dar en nuestro código.

**EXPLICACION**

<https://dev.to/juanmirod/tests-unitarios-en-javascript-con-jasmine-118e>