



Cómo realizar pruebas automatizadas en nuestras aplicaciones

PHP



Qualentum Lab

Cómo realizar pruebas automatizadas en nuestras aplicaciones

El testing juega un papel fundamental en la creación de aplicaciones web actual. Son imprescindibles para el desarrollo seguro, robusto y modular de cualquier aplicación. A lo largo de este fastbook, veremos qué se entiende por ‘testing’ en el desarrollo y cómo realizar diferentes tipos de pruebas en algunos de los frameworks más famosos de PHP.

¡Empezamos!

Autor: Jesús Donoso

- El testing para el desarrollo de aplicaciones sólidas
- PHPUnit: fundamentos y uso básico
- Los tipos principales de testing en PHP
- Test Driven Development (TDD)
- Conclusión
- Recursos extra de interés

El testing para el desarrollo de aplicaciones sólidas



El testing es esencial para comprender y mejorar la calidad del software desarrollado en cualquier lenguaje de programación.

En este apartado abordaremos diversos conceptos y prácticas que permiten verificar y validar el correcto funcionamiento de las aplicaciones. Pero comencemos entendiendo por qué la realización de pruebas es tan necesaria. Gracias al testing podemos garantizar la calidad, la confiabilidad y la eficiencia de las aplicaciones que estamos desarrollando. ¿Cuáles son sus funciones? En el siguiente listado recogemos las más relevantes:

1

Nos permite identificar y corregir errores en las primeras etapas del desarrollo, lo que reduce la probabilidad de problemas costosos en fases posteriores.

2

Contribuye a la creación de software robusto al garantizar que las funciones operen según lo esperado y cumplan con los requisitos establecidos.

3

Proporciona una red de seguridad al desarrollador al realizar cambios en el código, asegurando que las modificaciones no introduzcan nuevos errores o afecten negativamente a otras partes del sistema.

4

Reduce los riesgos asociados con el desarrollo al proporcionar una evaluación sistemática de la funcionalidad y seguridad de la aplicación.

5

Asegura que la aplicación se comporte de manera consistente a lo largo del tiempo y en diferentes entornos, proporcionando confianza tanto a los desarrolladores como a los usuarios finales. Los desarrolladores ganan confianza al implementar pruebas que validan el comportamiento esperado del software, y los usuarios finales confían más en un producto que ha sido sometido a pruebas exhaustivas.

6

Proporciona una base objetiva para la comunicación entre miembros del equipo de desarrollo, facilitando una comprensión compartida de los requisitos y expectativas en el código.

7

Aunque la implementación inicial de pruebas puede requerir tiempo adicional, a largo plazo acelera el desarrollo al evitar la acumulación de deudas técnicas y reducir el tiempo dedicado a la depuración.

Beneficios del testing automatizado

El testing automatizado en el desarrollo de software implica **la creación y ejecución de pruebas sin la intervención manual constante**. Utiliza código y herramientas especializadas para llevar a cabo pruebas de manera sistemática y eficiente.

La automatización del testing es esencial en entornos de desarrollo ágil, contribuyendo a la entrega rápida y confiable de software de alta calidad.

Aunque ya hemos visto alguno de ellos, vamos a concentrar, a continuación, los beneficios que nos aporta a los desarrolladores:

- La detección temprana de errores.** La automatización permite ejecutar pruebas de manera rápida y sistemática, identificando errores en las etapas iniciales del desarrollo antes de que se conviertan en problemas costosos.
- El ahorro de tiempo y recursos.** Las pruebas automatizadas se ejecutan de manera eficiente sin intervención manual, lo que ahorra tiempo y recursos en comparación con pruebas manuales repetitivas.
- Una ejecución consistente.** La automatización garantiza una ejecución consistente de las pruebas en diferentes entornos y configuraciones, lo que contribuye a una evaluación más precisa de la calidad del software.
- La repetibilidad.** Las pruebas automatizadas pueden repetirse fácilmente, permitiendo la verificación constante del comportamiento del software durante el desarrollo y las actualizaciones.
- Una mayor cobertura.** La automatización facilita la realización de un conjunto más amplio y completo de pruebas, lo que mejora la cobertura de código y escenarios de uso.
- Una rápida retroalimentación.** La ejecución automática de pruebas proporciona retroalimentación inmediata a los desarrolladores, lo que les permite abordar problemas de manera rápida y eficiente.
- Facilita el testing de regresión.** Las pruebas automatizadas son ideales para realizar pruebas de regresión, asegurando que las nuevas actualizaciones no afecten negativamente a funcionalidades existentes.

- Integración continua (CI).** La automatización es esencial para la implementación exitosa de prácticas de integración continua, donde las pruebas se ejecutan automáticamente con cada cambio en el código.
- Mejora la confianza en el código.** La consistencia y fiabilidad de las pruebas automatizadas aumentan la confianza tanto de los desarrolladores como de los usuarios finales en la estabilidad del software.
- Facilita el desarrollo ágil.** La automatización es fundamental para el desarrollo ágil, permitiendo ciclos de desarrollo rápidos y entregas frecuentes sin sacrificar la calidad.
- Identificación de problemas de rendimiento.** Las pruebas automatizadas pueden incluir escenarios que evalúan el rendimiento del sistema, identificando problemas de manera temprana.
- Documentación viva.** Las pruebas automatizadas sirven como documentación viva, proporcionando una descripción funcional de cómo se espera que opere cada parte del sistema.

PHPUnit: fundamentos y uso básico



PHPUnit, desarrollado por [Sebastian Bergmann](#), se destaca como un framework de pruebas unitarias para PHP ampliamente adoptado en la comunidad. Su principal propósito es asegurar la calidad del código PHP mediante la creación y ejecución de pruebas automatizadas.

La **instalación** de PHPUnit se realiza típicamente mediante Composer, la herramienta de gestión de dependencias para PHP. El proceso puede llevarse a cabo con el siguiente comando:

```
composer require --dev phpunit/phpunit
```

Las pruebas unitarias en PHPUnit se definen en clases que heredan de la clase `PHPUnit\Framework\TestCase`. Los métodos de prueba deben comenzar con la palabra clave 'test'. Por ejemplo:

```
class MiPrueba extends PHPUnit\Framework\TestCase {
    public function testSuma() {
        $this->assertEquals(4, 2 + 2);
    }
}
```

Las pruebas pueden **ejecutarse desde la línea de comandos** mediante el siguiente comando:

```
vendor/bin/phpunit
```

PHPUnit busca automáticamente las clases de prueba en el directorio 'tests'. Adicionalmente, facilita la generación de informes de cobertura de código y, para obtener un informe en formato HTML, se puede usar el comando que mostramos a continuación:

```
vendor/bin/phpunit --coverage-html coverage
```

Assertions

PHPUnit ofrece una variedad de *assertions* que permiten a los desarrolladores verificar diferentes condiciones durante la ejecución de las pruebas unitarias. A continuación, se presentan algunos de los *assertions* más comunes:

- **assertEquals:** verifica que dos valores sean iguales.

```
$testCase->assertEquals(3, $miVariable);
```

- **assertTrue** y **assertFalse**: verifican que una expresión sea verdadera o falsa, respectivamente.

```
$testCase->assertTrue($condicion);
$testCase->assertFalse($otraCondicion);
```

- **assertNull** y **assertNotNull**: verifican si un valor es nulo o no es nulo.

```
$testCase->assertNull($variableNula);
$testCase->assertNotNull($otraVariable);
```

- **assertSame** y **assertNotSame**: verifican que dos variables sean idénticas o no idénticas (mismo objeto).

```
$testCase->assertSame($objetoA, $objetoB);
$testCase->assertNotSame($otroObjetoA, $otroObjetoB);
```

- **assertGreaterThan** y **assertLessThan**: verifican que un valor sea mayor o menor que otro.

```
$testCase->assertGreaterThan(5, $valor);
$testCase->assertLessThan(10, $otroValor);
```

- **assertArrayHasKey**: verifica que un array tenga una clave específica.

```
$testCase->assertArrayHasKey('clave', $miArray);
```

- **assertStringContainsString**: verifica que una cadena contenga otra cadena.

```
$testCase->assertStringContainsString('buscar', $cadena);
```

- **assertCount**: verifica que un array o una clase que implementa 'Countable' tenga un número específico de elementos.

```
$testCase->assertCount(3, $miArray);
```

- **assertInstanceOf** y **assertNotInstanceOf**: verifica si una variable es una instancia de una clase específica.

```
$testCase->assertInstanceOf(MiClase::class, $objeto);
$testCase->assertNotInstanceOf(OtraClase::class, $otroObjeto);
```

- **assertFileExists** y **assertFileNotExists**: verifica si existe un archivo o directorio.

```
$testCase->assertFileExists('/ruta/a/mi/archivo.txt');
$testCase->assertFileNotExists('/otra/ruta');
```

Estos son solo algunos ejemplos de assertions disponibles en PHPUnit. La elección del assertion adecuado depende de la condición específica que se desea verificar en las pruebas unitarias. La [documentación oficial de PHPUnit](#)

Los tipos principales de testing en PHP



Se ha visto la importancia del testing en el desarrollo de aplicaciones hoy en día.

Existen varios tipos de pruebas, sin embargo, las tres que se comentan a continuación son las más comunes:

PRUEBAS UNITARIAS

PRUEBAS FUNCIONALES

PRUEBAS DE INTEGRACIÓN

Se centran en verificar unidades individuales de código, como métodos o funciones, de manera aislada.

PRUEBAS UNITARIAS

PRUEBAS FUNCIONALES

PRUEBAS DE INTEGRACIÓN

Evalúan el comportamiento de la aplicación desde la perspectiva del usuario.

PRUEBAS UNITARIAS

PRUEBAS FUNCIONALES

PRUEBAS DE INTEGRACIÓN

Se centran en verificar la interacción correcta entre diferentes partes o componentes del sistema. Estas pruebas pueden abordar cómo los diversos módulos interactúan entre sí.

En este apartado vamos a aprender cómo implementar pruebas en Symfony, Laravel y Codeigniter, nuestros tres frameworks de desarrollo 'estrella' en PHP

Testing en Symfony

Symfony integra un sistema robusto de testing que abarca varios tipos de pruebas para asegurar la calidad y confiabilidad del código.

Esta herramienta organiza las pruebas en la carpeta tests/dentro del directorio del proyecto, permitiendo una estructura clara y ordenada para diferentes tipos de pruebas.

Se apoya en PHPUnit como su framework principal para la escritura y ejecución de pruebas. Por lo tanto, los desarrolladores pueden aprovechar las funcionalidades de PHPUnit para realizar pruebas unitarias y funcionales.

1

Pruebas unitarias en Symfony

Es muy habitual utilizar PHPUnit para escribir pruebas unitarias en Symfony. Para entender cómo trabajar con estas pruebas, observa, en el siguiente ejemplo, la lógica de una clase MiClaseTest que extiende de TestCase. Dentro de la clase se pueden implementar tantas funciones como se consideren necesarias para el testeo de la aplicación.

```
class MiClaseTest extends \PHPUnit\Framework\TestCase
{
    public function testAlgo()
    {
        // Lógica de la prueba unitaria
    }
}
```

2

Pruebas funcionales en Symfony

Para la realización de este tipo de pruebas en Symfony es de gran utilidad la herramienta 'Symfony Panther'. Se trata de una herramienta que facilita la interacción con la interfaz de usuario y la simulación de solicitudes HTTP.

A continuación, observa la lógica de una clase *MiPruebaFuncional* que extiende de la clase *WebTestCase*. Dentro de la clase se pueden implementar funciones para probar cómo respondería la aplicación a un uso de un usuario final.

```
class MiPruebaFuncional extends
\Symfony\Bundle\FrameworkBundle\Test\WebTestCase
{
    public function testInteraccionUsuario()
    {
        $client = static::createPantherClient();
        // Lógica de la prueba funcional con interacción de usuario
    }
}
```

3

Pruebas de integración en Symfony

Las pruebas de integración se centran en verificar la interacción correcta entre diferentes partes o componentes del sistema. Estas pruebas pueden abordar cómo los diversos módulos interactúan entre sí.

```
class MiPruebaDeIntegracion extends
\Symfony\Bundle\FrameworkBundle\Test\KernelTestCase
{
    public function testIntegracionComponentes()
    {
        // Lógica de la prueba de integración
    }
}
```

Estos son solo algunos ejemplos de los tipos de pruebas que se pueden realizar en Symfony. Como ya sabes, deberás elegir el tipo de prueba en función de los aspectos específicos del sistema que deseas evaluar y asegurar. Symfony proporciona herramientas y estructuras que hacen que la implementación de nuestro test sea ágil y eficiente.

Testing en Laravel

Laravel simplifica el proceso de realización de pruebas al integrar PHPUnit y proporcionar un conjunto de herramientas específicas para pruebas unitarias, funcionales y de integración.

En los apartados, a continuación, se detalla cómo llevar a cabo diferentes tipos de pruebas en Laravel.

1

Pruebas unitarias en Laravel

En Laravel, a diferencia de Symfony, se pueden generar clases para la implementación de pruebas a través de **comandos artisan**. Para las pruebas unitarias, el comando que te vamos a mostrar genera una clase en el directorio *tests/Unit*:

```
php artisan make:test MiClaseTest --unit
```

La ejecución del comando genera un archivo *MiClaseTest.php* en el directorio *tests/Unit*. La nueva clase creada extiende la clase *TestCase* de *PHPUnit*. Pues bien, en su implementación se puede hacer uso de los métodos de aserción de *PHPUnit*.

```
// Ejemplo: tests/Unit/MiClaseTest.php
class MiClaseTest extends \Tests\TestCase
{
    public function testMetodo()
    {
        // Lógica de la prueba unitaria
        $this->assertTrue(true);
    }
}
```

Para ejecutar las pruebas unitarias configuradas basta con hacer uso del comando *artisan* de esta manera:

```
php artisan test
```

2

Pruebas funcionales y de integración en Laravel

En Laravel las pruebas funcionales y de integración comparten similitudes y los términos pueden ser intercambiados. Sin embargo, es importante tener en cuenta que, por un lado, las pruebas funcionales suelen enfocarse en la funcionalidad global de la aplicación desde la perspectiva del usuario, es decir, simular acciones que podría realizar un usuario. Por otro lado, **las pruebas de integración tienden a centrarse en evaluar que los diferentes componentes de la aplicación interactúan correctamente**. Estas pruebas pueden simular similitudes HTTP o la interacción con una base de datos.

Para generar una clase para la implementación de una prueba funcional o de integración en Laravel basta con ejecutar el comando *artisan*, tal y como mostramos aquí:

```
php artisan make:test MiPruebaDeIntegracion
```

La ejecución del comando genera un archivo *MiPruebaDeIntegracion.php* en el directorio *tests/Feature*. La nueva clase de la clase *TestCase* de PHPUnit. A continuación, se puede observar un ejemplo de una prueba de integración con un método asertivo y una solicitud HTTP.

```
// Ejemplo: tests/Feature/MiPruebaDeIntegracion.php
class MiPruebaDeIntegracion extends \Tests\TestCase
{
    public function testInteraccionConLaAplicacion()
    {
        $response = $this->get('/ruta');
        $response->assertStatus(200);
    }
}
```

Se puede hacer uso del siguiente comando *artisan* para ejecutar las pruebas funcionales o de integración implementadas:

```
php artisan test --feature
```

Laravel proporciona una serie completa de herramientas para realizar pruebas, y la documentación oficial ofrece recursos detallados y ejemplos adicionales. Accede a ella desde aquí este enlace: [testing en Laravel](#).

CodeIgniter simplifica la realización de pruebas a través de su biblioteca de pruebas unitarias integrada.

Aunque no tiene una integración directa con PHPUnit como Symfony o Laravel, proporciona herramientas para realizar pruebas bastante eficientes.

En Codeigniter las clases de prueba extienden de la clase TestCase y para ejecutarlas basta con hacer uso del siguiente comando:

```
vendor/bin/phpunit
```

En el fragmento de código que te compartimos a continuación se puede observar un ejemplo de prueba unitaria. También recuerda que se pueden realizar aserciones y verificar el comportamiento esperado de las funciones:

```
// Ejemplo: tests/MiModelo_test.php
class MiModelo_test extends TestCase
{
    public function testMetodo()
    {
        // Lógica de la prueba unitaria
        $resultado = funcion_que_quieres_probar();
        $this->assertEquals($resultado, 'valor Esperado');
    }
}
```

Ahora vamos a ver cómo se ha creado una clase llamada *MiControlador_test* siguiendo la convención de nombres *NombreControlador_test.php* con un ejemplo.

```
// Ejemplo: tests/MiControlador_test.php
class MiControlador_test extends TestCase
{
    public function testMetodo()
    {
        // Lógica de la prueba del controlador
        $output = $this->request('GET', 'mi_controlador/mi_metodo');
        $this->assertContains('Texto esperado', $output);
    }
}
```

En este caso la lógica implementada está pensada para simular solicitudes y verificar la respuesta generada por el controlador de la aplicación.

Pues bien, visto el ejemplo, me gustaría insistir en que, a pesar de no tener una integración completa con PHPUnit, CodeIgniter nos proporciona un entorno adecuado para realizar pruebas efectivas, aprovechando su biblioteca de pruebas unitarias y otras herramientas incorporadas.

Al igual que en los anteriores framework, recomiendo revisar la documentación oficial de CodeIgniter ya que nos ofrece información detallada sobre la realización de pruebas, algunos ejemplos y recomendaciones de buenas prácticas. Puedes acceder a ella desde el siguiente enlace: [testing en CodeIgniter](#).

Test Driven Development (TDD)



El desarrollo dirigido por pruebas en PHP (TDD por sus siglas en inglés) es una metodología de desarrollo de software que se enfoca en la escritura de pruebas automatizadas antes de escribir el código de producción.

A través del ciclo 'Red-Green-Refactor', los desarrolladores siguen un proceso iterativo que comienza con la escritura de una prueba que inicialmente falla (estado rojo), seguida de la implementación del código mínimo necesario para que la prueba pase (estado verde) y, finalmente, la refactorización del código para mejorarlo sin cambiar su comportamiento.

Veamos ahora un ejemplo más detallado de TDD en PHP con una clase que vamos a denominar 'Calculadora'. En primer lugar, hay que implementar la clase *CalculadoraTest* que contiene una prueba que verifica la suma de dos números:

```
// Ejemplo de prueba (PHPUnit - tests/CalculadoraTest.php)
class CalculadoraTest extends PHPUnit\Framework\TestCase
{
    public function testSuma()
    {
        $calculadora = new Calculadora();
        $resultado = $calculadora->sumar(2, 3);
        $this->assertEquals(5, $resultado);
    }
}
```

En esta fase, la ejecución de la prueba `testSuma` fallaría, porque la clase 'Calculadora' y su método `sumar` aún no están implementados. Así que, a continuación, implementamos la funcionalidad mínima necesaria para que la prueba resulte satisfactoria:

```
// Ejemplo de código (src/Calculadora.php)
class Calculadora
{
    public function sumar($a, $b)
    {
        $suma = $a + $b
        return $suma;
    }
}
```

Después de esta implementación mínima, la ejecución de la prueba `testSuma` debería ser satisfactoria. Y, finalmente, si es necesario, se puede realizar la refactorización del código probando que el resultado de la prueba continúa siendo satisfactorio. Por ejemplo, podría mejorarse la implementación del método `sumar` sin cambiar su comportamiento:

```
// Ejemplo de código refactorizado (src/Calculadora.php)
class Calculadora
{
    public function sumar($a, $b)
    {
        return $a + $b;
    }
}
```

Qué nos aporta TDD

El desarrollo dirigido por pruebas ofrece **diversos beneficios** que impactan positivamente en la calidad y eficiencia del desarrollo de software, aquí vamos a destacar los siguientes:



Detección temprana de errores. Al seguir la metodología TDD, los desarrolladores podemos identificar y corregir errores en las primeras etapas del desarrollo, minimizando la probabilidad de que se introduzcan defectos en el código final.



Documentación automática. Las pruebas escritas durante el proceso de TDD actúan como documentación automatizada, describiendo de manera clara y específica el comportamiento esperado del código. Esta práctica facilita la comprensión y el mantenimiento del código a medida que evoluciona.



Desarrollo incremental. TDD fomenta un enfoque incremental en el desarrollo, donde las funcionalidades se implementan paso a paso, cada una respaldada por pruebas específicas. Esta aproximación facilita la gestión del progreso y la entrega de software funcional en iteraciones más pequeñas.



Mejora de la mantenibilidad. Las pruebas sirven como 'salvaguardas' automáticas. Cuando se realizan modificaciones en el código, las pruebas aseguran que las funcionalidades existentes sigan siendo coherentes, simplificando la detección y corrección de problemas antes de que se conviertan en vulnerabilidades o defectos más complejos.



Confianza en el refactorizado. TDD proporciona una red de seguridad para la refactorización del código. Los desarrolladores pueden realizar cambios en la estructura o implementación del código con la certeza de que las pruebas detectarán posibles problemas.



Aumento de la eficiencia. A lo largo del tiempo, TDD puede aumentar la eficiencia del desarrollo al reducir el tiempo dedicado a la depuración y corrección de errores. La identificación temprana de problemas permite un desarrollo más fluido y rápido.



Facilita la colaboración. Las pruebas actúan como especificaciones objetivas del comportamiento esperado del código. Esto facilita la colaboración entre miembros del equipo, ya que todos pueden comprender y confiar en la funcionalidad del código basándose en las pruebas.



Reducción del miedo al cambio. TDD reduce el temor al cambio proporcionando una base sólida de pruebas que aseguran la estabilidad del código. Los desarrolladores pueden realizar modificaciones con confianza, ya que las pruebas indicarán si algo se ha roto.

En resumen, TDD en PHP es nuestro seguro de vida del código, es decir, nos brinda una mayor confianza en la calidad de este a lo largo del tiempo.

Conclusión



A modo de resumen, deberíamos destacar que las técnicas y herramientas de testing en el desarrollo de aplicaciones son indispensables hoy día para garantizar la calidad, la funcionalidad y la seguridad de los desarrollos. A través de la implementación y ejecución de pruebas, logramos identificar y corregir posibles errores y fallos, asegurando así una experiencia óptima para los usuarios finales.

Este proceso no solo contribuye a la reputación de la aplicación, sino que también minimiza costos asociados con posibles reparaciones. En última instancia, la inversión en pruebas rigurosas se traduce en un desarrollo más robusto y confiable, fundamental para el éxito a largo plazo de cualquier aplicación.

Recursos extra de interés



Qualentum Lab

Aquí te volvemos a compartir lecturas que deberías revisar para mejorar tu aprendizaje y como documentación de consulta:

- Documentación oficial PHPUnit:
<https://phpunit.de/documentation.html>.
- Documentacion oficial testing Laravel:
<https://laravel.com/docs/10.x/testing>.
- Documentación oficial testing Codeigniter:
https://codeigniter.com/userguide3/libraries/unit_testing.html.

¡Enhorabuena! Fastbook superado



Qualentum.com