Phpunit

los archivos de prueba tendrán el mismo nombre que la clase que queremos testear agregando el nombre test. Ej. UserTest.php

Nota1: Para que detecte y automatice correctamente los test debemos agregar el sufijo: **Test** a la clase. En el ejemplo anterior NO valdría: Usertest.php sino: UserTest.php Ahora veremos una clase de ejemplo que crearíamos para testear User

Nota2: Si estamos con laravel podemos crear las clases de test con un comando artisan:

```
php artisan make:test UserTest.php
```

(nos creará la clase en: Feature. Si se quiere en una carpeta: Unit agregar la opción: --unit)

```
use PHPUnit\Framework\TestCase;

class UserTest extends TestCase

{

    public function test_render()

    {

        require “User.php”;

        $user = new User(“mi usuario”);

        $expected = “lo que esperamos que aparezca”;

        $this->assertEquals($user→render(),$expected);

    }

}
```

Creamos un nuevo usuario, establecemos el valor que esperamos que salga y utilizamos: assertEquals() para comparar el resultado del método render() de la clase User que es lo que estamos testeando con el resultado esperado. Hay más funciones aparte de assertEquals() investigar por otras comparaciones

Para probar los test, debemos lanzar desde consola, con la ruta donde tengamos phpunit:

```
vendor/bin/phpunit $UserTest.php
```

Nos mostrará en consola si supera el test. Cada ok es una verificación correcta.

De nuevo, en laravel tenemos un comando artisan para ejecutar todos los test:

```
php artisan test
```

Recordar la importancia de programar con los test realizados. Observar que si pasas siempre los test a tu proyecto si tocas algún código para hacer compatible con el código nuevo que estás haciendo y has dejado mal algo que antes tenías bien te va a avisar y así tu proyecto llegará en buen estado al final

Tomemos la clase: Complejo:

```
class Complejo{
	protected $real=0;
	protected $img=0;

	function suma(Complejo $d):Complejo{
		$r = new Complejo();
		$r->setReal($this->real + $d->real)
			->setImg($this->img + $d->img);
		return $r;
	}

	function opuesto():Complejo{
		$r = new Complejo();
		$r->setReal(-$this->real )
			->setImg(-$this->img);
		return $r;
	}
```

```
        function resta():Complejo{
                return $this->suma($this->opuesto());
        }
```

La clase ComplejoTest mostrando un test para la suma():

```
use PHPUnit\Framework\TestCase;

class ComplejoTest extends TestCase
{
        public function test_suma()
        {
                require "complejo.php";
                $complejo1 = new Complejo();
                $complejo1->setReal(2)->setImg(3);

                $complejo2 = new Complejo();
                $complejo2->setReal(1)->setImg(4);

                $expected = (new Complejo())->setReal(3)->setImg(7);

                $this->assertEquals($complejo1->suma($complejo2),$expected);
                $this->assertEquals($complejo2->suma($complejo1),$expected);
        }
}
```

Test con bases de datos

Si estamos con laravel y eloquent hay unas formas específicas (sencillas) para hacer los test. Pero vamos a ver opciones que valgan también para PHP sin framework

Usaremos PDO. Así, ejecutaremos el correspondiente: new PDO() con los parámetros de la base de datos, usuario y contraseña para PHP sin framework. En Laravel usaremos: DB::getPdo(). En ambos casos vamos a trabajar con una base de datos en memoria para las pruebas (sqlite)

Así para PHP sin framework:

```
$pdo = new PDO('sqlite::memory:');
```

Con Laravel:

```
$pdo = DB::getPdo();
```

Para que la linea laravel funcione modificamos y dejamos sin comentar, las líneas para la base de datos en memoria del fichero: phpunit.xml (en la carpeta raíz del proyecto)

```
<env name="DB_CONNECTION" value="sqlite"/>
<env name="DB_DATABASE" value=":memory:"/>
```

Que originalmente se ven así:

```
<php>
    <env name="APP_ENV" value="testing"/>
    <env name="BCRYPT_ROUNDS" value="4"/>
    <env name="CACHE_DRIVER" value="array"/>
    <!-- <env name="DB_CONNECTION" value="sqlite"/> -->
    <!-- <env name="DB_DATABASE" value=":memory:"/> -->
    <env name="MAIL_MAILER" value="array"/>
    <env name="QUEUE_CONNECTION" value="sync"/>
    <env name="SESSION_DRIVER" value="array"/>
    <env name="TELESCOPE_ENABLED" value="false"/>
</php>
```

Veamos un ejemplo de test de una clase DAO:

```
class AsignaturaDAOTest extends TestCase{

	public $databaseCreated = false;

	public function setUp(): void{
		parent::setUp();
		if(! $this->databaseCreated ){
			$pdo = DB::getPdo();
			require 'CreateDatabase.php';
			$this->databaseCreated = true;
		}
	}

	public function test_1_findAll(): void {
		$pdo = DB::getPdo();

		$asignaturaDAO = new AsignaturaDAO($pdo);
		$asignaturas = $asignaturaDAO->findAll();
		assertTrue(count($asignaturas) == 8);
	}
}
```

Tenemos que crear la base de datos en memoria antes de la ejecución de los test (luego al acabar desaparece) Se pensaría que sería apropiado: setUpBeforeClass() para crear en ese momento la base de datos (se ejecuta una única vez antes de ejecutar la clase de test) Pero hay muchas cosas que hace Laravel internamente hasta que esté todo operativo. Usaremos en su lugar la función: setUp() que se ejecuta previamente a cada uno de los test y hacemos que se ejecute la creación una única vez consultando un booleano: `if(! $this->databaseCreated )`.

La línea: `require 'CreateDatabase.php';` lo que hace es cargar un fichero que ejecuta sentencias exec pdo ($pdo→exec()) para ir creando la base de datos. Veamos un ejemplo de ese fichero CreateDatabase.php donde se crea una tabla alumnos y una tabla asignaturas:

```
<?php
$pdo->exec("
create table alumnos(
    dni CHARACTER(20),
    nombre CHARACTER(50),
    apellidos CHARACTER(50),
    fechanacimiento BIGINT,

    CONSTRAINT pk_alumnos PRIMARY KEY(dni)
)
");
$pdo->exec("
create table asignaturas(
    id int AUTO_INCREMENT,
    nombre CHARACTER(50),
    curso CHARACTER(50),
    CONSTRAINT pk_asignaturas PRIMARY KEY(id),
    CONSTRAINT uc_nombrecurso UNIQUE(nombre,curso)
)
");
?>
```

Test ordenados

Finalmente, en el código de test aparecen los métodos a ejecutar escritos con el prefijo: **test_1_** Esto es únicamente porque si queremos que se ejecuten de forma ordenada los test, se puede especificar mediante un orden alfabético. Así, si todos los llamamos con el prefijo: test_num_ los ejecutaría de forma ordenada