Norbert Kiraly

Gazdaság informatika|

Webes alkalmazások tesztelése

Biztosak vagyunk benne, hogy hallott arról a fogalomról, hogy “bug” amikor a webes alkalmazásokról beszélünk. Mondatok mint például :”Találtunk egy bugot (hibát) az alkalmazásban ami...” amit követ pár nemkívánatos viselkedés, gyakoribbak, mint az ön hinné.A kód megírása nem az egyetlen feladata a feljlesztőnek;a tesztelés is kritikus része.Nem kellene kiadnia olyan verzióját az alkalmazásának amelyik nem volt tesztelve.Azonban el tudná képzelni, hogy minden egyes sor átirásánál tesztelje az egész alkalmazást? Egy rémálom lenne!

Nem mi vagyunk az elsők akiknek van ez a problémája, szóval szerencsére a fejlesztők találtak egy elég jó megoldást erre a problémára. Valójában több mint, egy megoldást találtak ami miatt a tesztelés egy elég fontos téma. Még a teszt fejlesztő is egy gyakori szerep lett. Ebben a fejezetben bevezetünk a kódja tesztelésének egyik megközelítésébe: egységtesztelés(unit teszt).

Ebben a fejezetben tanulni fog:

* Hogyan müködnek az egységtesztek(unit test)
* PHPUnit beállítása a kódja teszteléséhez
* Tesztek írása állításokkal,adatszolgáltatókkal, álcaobjektumokkal
* Jó és rossz praktikák egységtesztelésnél

# **A tesztek szükségessége**

Amikor egy projekten dolgozik nagy az esélye, hogy nem ön az egyetlen fejlesztő, aki dolgozik majd ezzel a kóddal. Még abban az esetben is ha Ön az egyetlen aki valaha is változtatni fog rajta,ha ezt néhány héttel a létrehozás után teszi meg, akkor valószínüleg nem fog emlékezni az összes helyre, amelyre ez a kódreszlet hatással van. Oké, tegyük fel, hogy ön az egyetlen fejlesztő, és a memóriád is határtalan; vajon képes lenne-e ellenőrizni, hogy egy gyakran használt objektumon, például egy kérésen végzett módosítás mindig működni fog-e a várt módón? Sőt ami még fontosabb, minden egyes alkalommal megtenné ezt amikor egy apró változtatást végez?

## Teszttipusok

Az alkalmazás írása, a meglévő kód módosítása, vagy új funkciók hozzáadása, nagyon fontos, hogy jó visszajelzést kapjunk.Honnan tudja, hogy a visszajelzés eleg jó? Meg kell valósítania az AIEOU alapelveket:

* **Automatikus:** A visszajelzés megszerzése a lehető legfájdalommentesebben történjen.Egyetlen parancs futtatásával mindig előnyösebb, mintha manuálisan kellene tesztelni az alkalmazást.
* **Kiterjedt:** A lehető legtöbb felhasználási esetet le kell tudnunk fedni, beleértve azokat a szélsőséges eseteket amelyeket nehéz előre látni a kód írása során.
* **Azonnali:** A lehető leghamarabb meg kell kapnia önnek. Ez azt jelenti, hogy a visszajelzés amit közvetlenül a változtatás bevezetése során kapunk, sokkal jobb, mint a visszajelzés, amit azután kapunk, hogy a kódunk már gyártásban van.
* **Nyitott:** Az eredményeknek átláthatónak kell lenniük, és a teszteknek a következőket kell nyújtaniuk: betekintést más fejlesztők számára, hogy hogyan integrálják vagy működtessék a kódot.
* **Hasznos:** Olyan kérdésekre kell választ adniuk mint például: “Fog-e működni ez a változtatás?”, „Nem szakítsa-e meg az alkalmazás működését?”, vagy “Van-e olyan szélsőséges eset, amely nem működik megfelelően?”.

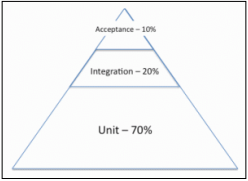
Tehát, ha a koncepció kezdetben elég furcsa is a kódja tesztelésének legjobb módja...még több kód. Pontosan! Kódot fogunk írni azzal a céllal, hogy teszteljük az alkalmazásunk kódját. Hogy miért? Nos, ez a legjobb módja annak, hogy kielégítsük az összes AEIU elvet, és a következő előnyökkel jár:

* A teszteket egyetlen parancs futtatásával tudjuk végrehajtani a parancssorból vagy akár a kedvenc integrált fejlesztési környezetünkből(IDE). Nincs szükség arra, hogy manuálisan teszteljük az alkalmazást böngészőn keresztűl folyamatosan.
* A tesztet csak egyszer kell megírnunk. Kezdetben ez egy kicsit fájdalmas lehet, de ha egyszer megírtuk a kódot, nem kell majd újra és újra megismételnünk. Ez azt jelenti, hogy némi munka után képesek leszünk minden egyes esetet tesztelni könnyedén. Ha kézzel kellene tesztelnünk, az összes használati esettel együtt és az összes szélsőséges esettel együtt az egy rémálom lenne.
* Nem kell, hogy az egész alkalmazás működjön ahhoz, hogy tudja, hogy a kódja működik-e. Képzelje el, hogy megírja a routerét: annak érdekében hogy megtudja, működik-e, meg kell várnia, amíg az alkalmazás működik a böngészőben. Ehelyett megírhatja a tesztjeit, és lefuttathatja őket, amint befejezte az osztályát.
* A tesztek írása során visszajelzést kap arról, hogy mi az, ami nem sikerül. Ez nagyon hasznos, ha tudni szeretné, hogy a router egy bizonyos funkciója nem működik, és mi az oka a hibának, ami jobb, mint kapni egy 500-as hibát a böngészönkben.

Reméljük, hogy mostanra már meggyőztük Önt arról, hogy a tesztek írása nélkülözhetetlen. Ez volt azonban a könnyű rész. A probléma az, hogy számos különböző megközelítéseket tudnunk. Írunk-e olyan teszteket, amelyek az egész alkalmazást tesztelik, vagy olyan teszteket, amelyek bizonyos részeket tesztelnek? Elszigeteljük a tesztelt területet a többitől? Akarunk-e interakcióba lépni az adatbázissal vagy más külső erőforrásokkal a tesztelés során? A válaszaitól függően dönti el, hogy milyen típusú teszteket szeretne írni. Beszéljük meg a három fő megközelítést, amelyekkel a fejlesztők egyetértenek:

* **Egységtesztek**: Ezek olyan tesztek, amelyeknek nagyon fókuszált a hatókörük. Céljuk egyetlen osztály vagy metódus tesztelése, elszigetelve azokat a kód többi részétől. Vegyük a Sale domain osztályt példának: van néhány logikai eleme a hozzáadással kapcsolatban. könyvek hozzáadásával kapcsolatban, igaz? Egy egységteszt egyszerűen csak egy új Sale-t példányosít, könyveket ad hozzá az objektumhoz, és ellenőrzi, hogy a könyvek tömbje érvényes-e. Az egységtesztek szupergyorsak csökkentett hatókörüknek köszönhetően, így több különböző forgatókönyvet is létrehozhat az ugyanazon funkcionalitás többféle szcenárióját, amelyek lefedik az összes elképzelhető szélsőséges esetet. Ezek az izoláltak is, ami azt jelenti, hogy nem fog minket túlságosan érdekelni, hogy az alkalmazásunk minden darabja integrálva van. Ehelyett gondoskodni fogunk arról, hogy minden egyes darab tökéletesen működik.
* **Integrációs tesztek**: Ezek szélesebb körű tesztek.Céljuk annak ellenőrzése, hogy az alkalmazás minden darabja eggyütt működik-e, ezért a hatáskörük nincs korlátozva egy osztályra vagy funkcióra, hanem inkább osztályok egy csoportjára vagy az egész alkalmazásra terjed ki. Még mindig van némi elszigeteltség arra az esetre, ha nem akarunk egy valódi adatbázist, vagy valamilyen más külső webes szolgáltatástól függünk. Egy példa a mi alkalmazásunkban az lenne, ha szimulálnánk egy Request objektumot, elküldenénk a routerhez, és ellenőrizzük, hogy a válasz megfelel-e a várakozásoknak.
* **Elfogadási tesztek**: Ezek még szélesebb körű tesztek. A következőket próbálják tesztelni a teljes funkcionalitást a felhasználó szemszögéből. A webes alkalmazásoknál ez azt jelenti, hogy elindítunk egy böngészőt, és szimuláljuk a felhasználó által végrehajtott kattintásokat. a böngésző válaszát minden egyes alkalommal ellenőrizve. És igen, mindezt kódon keresztül! Ezek a tesztek lassabban futtathatók, ahogy azt el tudja képzelni, mert a hatókörük nagyobb, és a böngészővel való munka lelassítja őket. eléggé lelassítja.

Tehát, ezek közül a tesztek közül melyiket írja meg? A válasz az összes mindet. A trükk az, hogy tudja, mikor és hányat kell írnia az egyes típusokból. Egy jó megközelítés, ha sok egységtesztet ír, amelyek abszolút mindenre kiterjednek a kódban, majd kevesebb integrációs tesztet írunk, hogy megbizonyosodjunk arról, hogy az alkalmazása minden összetevője együtt működik, és végül az elfogadási tesztek írása, de csak a következő elemeket teszteli az alkalmazás fő folyamatait. Az alábbi tesztpiramis ezt az elképzelést reprezentálja:



Az ok egyszerű: az igazi visszajelzés az egységtesztekből fog érkezni. Ezek megmondják, ha valamit elrontottál a változtatásokkal, amint befejezte az írást, mert az egységtesztel végrehajtása egyszerű és gyors. Ha már tudja, hogy az összes osztálya és a függvények a várt módon viselkednek, meg kell győződnie arról, hogy együtt tudnak működni. Ehhez azonban nem kell újra tesztelnie az összes szélsőséges esetet; ezt már megtette az egységtesztek írásakor Itt csak néhány integrációs tesztet kell írnia, amelyek megerősítik, hogy az összes darab megfelelően kommunikál. Végül, hogy megbizonyosodjunk arról, hogy nem csak hogy a kód működik, hanem a felhasználói élmény is megfelelő. Írunk egy elfogadási tesztet, amely egy felhasználót utánoznak, aki végigmegy a különböző nézeteken. Itt a tesztek nagyon lassúak és csak akkor lehetségesek, ha a folyamat már befejeződött, így a visszajelzés később jön. Elfogadási teszteket fogunk hozzáadni, hogy megbizonyosodjunk arról, hogy a fő áramlások működnek, de nem kell minden egyes forgatókönyvet tesztelnünk, mivel ezt már megtettük az integrációs ésegységtesztekkel.

## Egységtesztek és kódlefedettség

Most, hogy már tudja, mik azok a tesztek, miért van rájuk szükségünk, és milyen típusú tesztek vannak. A fejezet hátralévő részében a jó egységtesztek írására fogunk összpontosítani, mivel ezek lesznek azok, amelyek a legtöbb időt lekötik majd.

Ahogy korábban már elmagyaráztuk, a egységtesztek lényege, hogy megbizonyosodjunk arról, hogy egy kódrészlet, általában egy osztály vagy metódus, az elvárásoknak megfelelően működik. Mivel egy metódus kódmennyisége alacsony, a teszt futtatása szinte alig vesz igénybe időt. Ezt kihasználva, több tesztet fogunk futtatni, és megpróbálunk minél több felhasználási esetet lefedni, amennyit csak lehet.

Ha nem most hall először az egységtesztekről, akkor talán ismeri a **kódlefedettség** fogalmát. Ez a fogalom a tesztjeink által végrehajtott kód mennyiségére utal, azaz a tesztelt kód százalékos arányára. Például, ha az alkalmazásunknak 10 000 sora van, és a tesztjei összesen 7500 sort tesztelnek, a kódlefedettség 75%. Vannak eszközök, amelyek jeleket mutatnak a kódon, jelezve, hogy egy adott sor tesztelt vagy nem, ami nagyon hasznos annak megállapításához, hogy az alkalmazás mely részei nem teszteltek, és így figyelmezteti Önt, hogy veszélyesebb megváltoztatni őket.

A kódlefedettség azonban egy kétélű kard. Miért van ez így? Azért, mert a fejlesztők hajlamosak a kódlefedettség megszállottjaivá válni, és a 100%-os lefedettségre törekednek Azonban tisztában kell lennie azzal, hogy a kódlefedettség csak egy következmény, nem pedig a cél. A célja az, hogy olyan egységteszteket írjon, amelyek ellenőrzik az összes felhasználási esetet a kód bizonyos darabjaiban, hogy nagyobb biztonságban érezze magát minden alkalommal, amikor változtatnia kell ezen a kódon. Ez azt jelenti, hogy egy adott módszer esetében nem biztos, hogy elég egyetlen tesztet írni, mert ugyanaz a sor különböző bemeneti értékekkel eltérően viselkedhet. Ha azonban a kód lefedettségére összpontosít, egyetlen teszt megírása kielégítené azt, és nem biztos, hogy több teszt írására nincs szükség.

# **PHPUnit Integrálása**

A tesztek írása olyan feladat, amit egyedül is meg tud csinálni; csak olyan kódot kell írnia, ami kivételeket dob, ha a feltételek nem teljesülnek, majd futtassa a szkriptet bármikor, amikor amikor szüksége van rá. Szerencsére más fejlesztők nem elégedtek meg ezzel a kézi folyamattal, így olyan eszközöket vezettek be, amelyek segítségével automatizálhatjuk ezt a folyamatot, és jó visszajelzéseket kaphatunk. A PHP-ben leginkább használt eszköz a PHPUnit. A PHPUnit egy olyan keretrendszer, amely egy sor eszközt biztosít a tesztek könnyebb megírásához, lehetőséget ad a tesztek automatikus futtatására, és hasznos visszajelzést ad a fejlesztőnek.

A PHPUnit használatához hagyományosan a laptopunkra telepítettük. Ennek során, hozzáadtuk a keretrendszer osztályait, hogy tartalmazza a PHP elérési útvonalát és a futtatható állományt a tesztek futtatásához. Ez kevésbé volt ideális, mivel arra kényszerítettük a fejlesztőket, hogy telepítsenek a egy újabb eszközt a fejlesztőgépükre. Napjainkban a Composer (lásd a fejezetben 6, Adaptálás az MVC-hez, hogy felfrissítse az emlékezetét) segít nekünk a PHPUnit beépítésében függőségként a projektbe. Ez azt jelenti, hogy a Composer futtatásával, amelyet a a többi függőség megszerzéséhez biztosan meg kell tenned, a PHPUnit is elérhetővé válik. Adjuk hozzá, ezután a következőket a composer.json fájlba:

{

//...

"require":

{

"monolog/monolog": "^1.17",

"twig/twig": "^1.23"

},

"require-dev": {

"phpunit/phpunit": "5.1.3"

},

"autoload": {

"psr-4": {

"Bookstore\\": "src"

}

}

}

Vegye figyelembe, hogy ez a függőség require-dev-ként kerül hozzáadásra. Ez azt jelenti, hogy a függőség csak akkor fog letöltődni, amikor fejlesztői környezetben vagyunk, de nem lesz része annak az alkalmazásnak, amit a termelésben fogunk telepíteni, mivel nem kell futtatnunk a teszteket. A függőség megszerzéséhez, mint mindig, futtassuk a: composer update parancsot.

Egy másik megközelítés a PHPUnit globális telepítése, hogy az összes projektet az Ön fejlesztési környezetében használhatja, ahelyett, hogy minden egyes alkalommal helyileg telepítené. A az eszközök globális telepítéséről a Composer segítségével a https://akrabat.com/global-installation-of-php-tools-with-composer/ oldalon olvashat.

## A PHPUnit.xml fájl

PHPUnit-nak szüksége van egy phpunit.xml fájlra annak érdekében, hogy meghatározza, hogyan akarjuk futtatni a teszteket. Ez a fájl egy sor szabályt határoz meg, mint például, hogy hol vannak a tesztek, milyen kódot tartalmaznak a tesztek és így tovább. Adjuk hozzá a következő fájlt a gyökérkönyvtárunkhoz:

<?xml version="1.0" encoding="UTF-8"?>

<phpunit backupGlobals="false"

backupStaticAttributes="false"

colors="true"

convertErrorsToExceptions="true"

convertNoticesToExceptions="true"

convertWarningsToExceptions="true"

processIsolation="false"

stopOnFailure="false"

syntaxCheck="false"

bootstrap="vendor/autoload.php"

>

<testsuites>

<testsuite name="Bookstore Test Suite">

<directory>./tests/</directory>

</testsuite>

</testsuites>

<filter>

<whitelist>

<directory>./src</directory>

</whitelist>

</filter>

</phpunit>

Ez a fájl elég sok mindent definiál. A legfontosabbakat a következőkben ismertetjük:

* convertErrorsToExceptions, convertNoticesToExceptions és convertWarningsToExceptions igazra állítása esetén a tesztek sikertelenek lesznek, ha PHP hiba, figyelmeztetés vagy értesítés van. A cél az, hogy a kódod nem tartalmaz kisebb hibákat az szélsőséges esetekben, amelyek mindig a potenciális problémák forrása
* A stopOnFailure megmondja a PHPUnit-nak, hogy folytassa-e a többi teszt végrehajtását vagy sem, ha van egy sikertelen teszt. Ebben az esetben az összes tesztet le akarjuk futtatni. hogy megtudjuk, hány teszt sikertelen és miért.
* A bootstrap meghatározza, hogy melyik fájlt kell végrehajtanunk a tesztek futtatás megkezdése előtt. A legáltalánosabb használat az autoloader beépítése, de akár a olyan fájlt is tartalmazhat, amely inicializál néhány függőséget, például adatbázisokat vagy konfigurációs olvasók.
* A testsuites meghatározza azokat a könyvtárakat, ahol a PHPUnit teszteket keres. A mi esetünkben a ./tests-t definiáltuk, de hozzáadhatnánk még többet is, ha lenne más könyvtárakban.
* A whitelist azon könyvtárak listáját határozza meg, amelyek tartalmazzák azt a kódot, amelyet mi tesztelünk. Ez hasznos lehet a kódlefedettséggel kapcsolatos kimenet generálásához.

Amikor a teszteket a PHPUnit segítségével futtatjuk, csak győződjünk meg róla, hogy a következő parancsot futtatjuk le ugyanabból a könyvtárból, ahol a phpunit.xml fájl található. Megmutatjuk, hogyan kell a következő szakaszban.

## Az első tesztje

Rendben, elég volt az előkészületekből és az elméletből; írjunk egy kis kódot. Megírjuk teszteket az alap ügyfélhez, ami egy domain objektum kevés logikával. Először is, át kell alakítanunk az Unique tulajdonságot(trait), mivel még mindig tartalmaz néhány felesleges kódot, miután integráljuk az alkalmazásunkat a MySQL-lel. Arról a képességről beszélünk, hogy a a következő elérhető ID-t, amit most az autoincremental mező kezel. Távolítsuk el, a kódot a következőképpen hagyjuk meg:

<?php

namespace Bookstore\Utils

trait Unique{

protected $id

}

public function setId(int $id) {

$this->id = $id;

}

public function getId(): int {

return $this->id;

}

}

A tesztek a tests/ könyvtárban lesznek. A könyvtárak szerkezetének a következőnek kell lennie mint az src/ könyvtárban, hogy könnyebb legyen azonosítani, hogy az egyes teszteknek hol kell lenniük. A fájl- és osztályneveknek Test-re kell végződniük, hogy a PHPUnit tudja, hogy egy fájl teszteket tartalmaz. Ennek ismeretében a tesztünknek a tests/Domain/Customer/BasicTest.php könyvtárban kell lennie. fájlban kell lennie, a következőképpen:

<?php

namespace Bookstore\Tests\Domain\Customer;

use Bookstore\Domain\Customer\Basic;

use PHPUnit\_Framework\_TestCase;

class BasicTest extends PHPUnit\_Framework\_TestCase {

public function testAmountToBorrow() {

$customer = new Basic(1, 'han', 'solo', 'han@solo.com');

$this->assertSame(

3,

$customer->getAmountToBorrow(),

'Basic customer should borrow up to 3 books.'

);

}

}

Mint látható, a BasicTest osztály a PHPUnit\_Framework\_TestCase osztályból származik. Minden tesztosztálynak ebből az osztályból kell kiterjesztenie. Ez az osztály egy sor metódussal rendelkezik amelyek lehetővé teszik az állítások készítését. Az állítás a PHPUnitban nem más, mint egy elvégzett ellenőrzés egy értéken. Az állítások lehetnek más értékekkel való összehasonlítások, bizonyos értékek ellenőrzése, vagy a értékek attribútumainak ellenőrzése, és így tovább. Ha egy állítás nem igaz, a tesztet a következő jelöléssel látjuk el sikertelennek minősül, és a megfelelő hibaüzenet jelenik meg a fejlesztő számára. A példa az assertSame metódust használó állítást mutatja, amely két értéket hasonlít össze, elvárja, hogy mindkettő pontosan ugyanaz legyen. A harmadik argumentum egy hiba üzenet, amelyet az állítás sikertelenség esetén megjelenít.

Figyeljük meg azt is, hogy a test kezdetű függvénynevek azok, amelyeket a PHPUnit végez el. Ebben a példában egy egyedi tesztünk van, melynek neve testAmountToBorrow amely egy alap ügyfél példányosítását végzi, és ellenőrzi, hogy a kölcsönzött könyvek mennyisége, amelyet a ügyfél kölcsönözni tud, 3. A következő részben megmutatjuk, hogyan futtathatjuk ezt a tesztet, és kapjunk visszajelzést róla.

Opcionálisan bármilyen függvénynevet használhat, ha a @test megjegyzést hozzáadja a módszer DocBlockjába, az alábbiak szerint:

/\*\*

\* @test

\*/

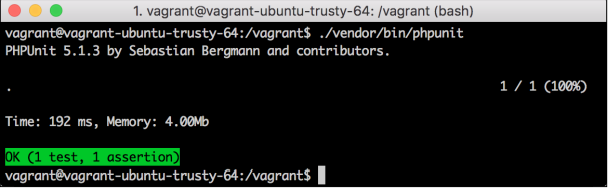
public function thisIsATestToo() {

//...

}

## Tesztek futtatása

A tesztek futtatásához a Composer által generált szkriptet kell futtatnia a vendor/bin könyvtárban. Ne feledje, hogy mindig a projekt gyökérkönyvtárából futassa, hogy a PHPUnit megtalálja a phpunit.xml konfigurációs fájlt.Ezt követően írja be a ./vendor/bin/phpunit parancsot.



A program végrehajtásakor megkapjuk a tesztek által adott visszajelzést. A kimenet azt mutatja, hogy egy teszt (egy metódus) és egy állítás van, és hogy a ezek kielégítőek voltak-e. Ezt a kimenetet szeretnénk látni minden egyes alkalommal, amikor futtatjuk a tesztjeinket, de több sikertelen tesztet fogunk kapni, mint amennyit szeretnénk. Vessünk egy pillantást a őket a következő teszt hozzáadásával:

public function testFail() {

$customer = new Basic(1, 'han', 'solo', 'han@solo.com');

$this->assertSame(

4,

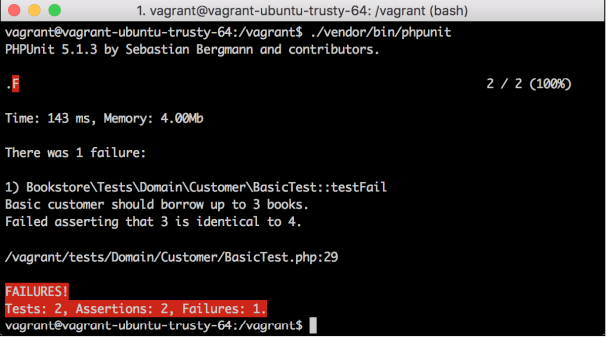
$customer->getAmountToBorrow(),

'Basic customer should borrow up to 3 books.'

);

}

Ez a teszt nem fog sikerülni, mivel azt ellenőrizzük, hogy a getAmountToBorrow 4-et ad-e vissza, de a tudja, hogy mindig 3-at ad vissza. Futtassuk le a teszteket, és nézzük meg, hogy milyen kimenetet kapunk.

 Gyorsan megállapíthatjuk, hogy a kimenet nem jó a piros szín miatt. Ez azt mutatja, hogy hogy hiba történt, és rámutat a hibás osztályra és tesztmódszerre. A visszajelzés rámutat a hiba típusára (mivel a 3 nem azonos a 4-gyel), és opcionálisan a hiba üzenetre, amelyet az assert metódus meghívásakor adtunk hozzá.

# **Egységtesztek írása**

Kezdjünk bele a PHPUnit által a tesztek írásához kínált összes funkcióba. Ezeket a funkciókat különböző alfejezetekre osztjuk: tesztek beállítása, állítások, kivételek és adatszolgáltatók. Természetesen nem kell mindezeket az eszközöket használni minden egyes alkalommal, amikor tesztet ír.

## A kezdete és vége a tesztnek

A PHPUnit lehetőséget ad arra, hogy egy közös forgatókönyvet állítson be minden egyes teszthez az osztályban. Ehhez a setUp metódust kell használnia, amely, ha van, minden egyes alkalommal amikor az osztály egy tesztje végrehajtásra kerül. Az osztály azon példánya, amelyik meghívja a setUp és a test metódusokat meghívó osztály ugyanaz, így az osztály tulajdonságait használhatja a a kontextus elmentéséhez. Az egyik gyakori felhasználás az lenne, hogy létrehozzuk azt az objektumot, amelyel a tesztjeinket használjuk, abban az esetben, ha ez mindig ugyanaz. Példaként írjuk a következő kódot a tests/Domain/Customer/BasicTest.php:

<?php

namespace Bookstore\Tests\Domain\Customer;

use Bookstore\Domain\Customer\Basic;

use PHPUnit\_Framework\_TestCase;

class BasicTest extends PHPUnit\_Framework\_TestCase {

private $customer;

public function setUp() {

$this->customer = new Basic(

1, 'han', 'solo', 'han@solo.com'

);

}

public function testAmountToBorrow() {

$this->assertSame(

3,

$this->customer->getAmountToBorrow(),

'Basic customer should borrow up to 3 books.'

);

}

}

Amikor a testAmountToBorrow meghívásra kerül, a $customer tulajdonság már inicializálva van a setUp metódus végrehajtása révén. Ha az osztálynak egynél több tesztje van, akkor a setUp metódus minden alkalommal végrehajtásra kerülne.

Bár ritkábban használják, van egy másik módszer is, amelyet a tisztításhoz használnak a teszt végrehajtása után: a tearDown. Ez ugyanígy működik, de a az osztály minden egyes tesztjének végrehajtása után hajtódik végre. Lehetséges felhasználási módok a következők: adatbázis-adatok, kapcsolatok lezárása, fájlok törlése stb.

## Állítások

Az állítások fogalmával már megismerkedtünk, ezért most csak felsoroljuk a leggyakoribbakat ebben a szakaszban. A teljes listáért javasoljuk, hogy látogasson el a hivatalos dokumentációt a https://phpunit.de/manual/6.5/en/appendixes.assertions.html oldalra. azonban, hogy őszinte legyek, valószínűleg sokukat nem fogjuk használni.

Az első állítás típus, amit látni fogunk, a Boolean állítás, azaz a amely azt vizsgálja, hogy egy érték igaz vagy hamis. A módszerek olyan egyszerűek, mint assertTrue és assertFalse, és egy paramétert várnak el, egy állítandó érték, és opcionálisan egy szöveg, amelyet sikertelenség esetén megjelenítünk. Ugyanabban a BasicTest osztályban a következő tesztet írjuk be:

public function testIsExemptOfTaxes() {

$this->assertFalse(

$this->customer->isExemptOfTaxes(),

'Basic customer should be exempt of taxes.'

);

}

Ez a teszt biztosítja, hogy az alapfogyasztó soha nem mentesül az adó alól. Megjegyezzük, hogy ugyanezt az állítást a következőkkel is megtehetnénk:

$this->assertSame(

$this->customer->isExemptOfTaxes(),

false,

'Basic customer should be exempt of taxes.'

);

Az állítások második csoportját az összehasonlító állítások alkotják. A leghíresebb az assertSame és az assertEquals. Az elsőt már használta, de az biztos abban, hpgy mit jelent? Adjunk hozzá egy másik tesztet, és futtassuk le:

public function testGetMonthlyFee() {

$this->assertSame(

5,

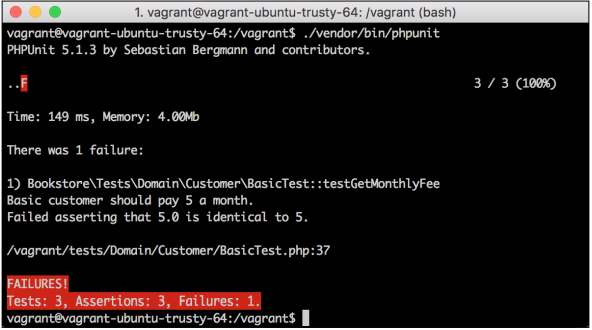
$this->customer->getMonthlyFee(),

'Basic customer should pay 5 a month.'

);

}

A teszt eredménye a következő képernyőképen látható:



A teszt nem sikerült! Ennek oka, hogy az assertSame egyenértékű az összehasonlítással, identitással, azaz típuszsonglőrködés nélkül. A getMonthlyFee módszer eredménye mindig egy lebegőszám, és mi egy egész számmal fogjuk összehasonlítani,tehát soha nem lesz ugyanaz, ahogy azt a hibaüzenet is jelzi.Változtassuk meg az álliítást assertEquals-ra ami egyenlőséget használva hasonlít, és a teszt mostmár ármegy.

Amikor objektumokkal dolgozunk, egy állítással ellenőrizhetjük, hogy egy adott objektum az elvárt osztály példánya-e vagy sem. Ilyenkor ne feledjük, hogy a teljes osztály nevét, mivel ez egy elég gyakori hiba. Még jobb, ha a osztály nevét a ::class használatával, például Basic::class. Adja hozzá a következő tesztet a tests/Domain/Customer/CustomerFactoryTest.php fájlba:

<?php

namespace Bookstore\Tests\Domain\Customer;

use Bookstore\Domain\Customer\CustomerFactory;

use PHPUnit\_Framework\_TestCase;

class CustomerFactoryTest extends PHPUnit\_Framework\_TestCase {

public function testFactoryBasic() {

$customer = CustomerFactory::factory(

'basic', 1, 'han', 'solo', 'han@solo.com'

);

$this->assertInstanceOf(

Basic::class,

$customer,

'basic should create a Customer\Basic object.'

);

}

}

Ez a teszt létrehoz egy ügyfelet az ügyfélgyár segítségével. Mivel az ügyfél típusa alapvető ügyfél, az eredménynek a Basic egy példányának kell lennie amit az assertInstanceOf metódussal tesztelünk. Az első argumentum a várt osztály, a második az objektum amit tesztelünk, a harmadik pedig a hibaüzenet. Ez a teszt segít nekünk abban is, hogy megjegyezzük az objektumokkal való összehasonlító állítások viselkedését. Hozzunk létre egy alapvető ügyfél objektumot az elvártaknak megfelelően, és hasonlítsuk össze a gyár eredményével. Ezután futtassuk le a tesztet, az alábbiak szerint:

$expectedBasicCustomer = new Basic(1, 'han', 'solo', 'han@solo.com');

$this->assertSame(

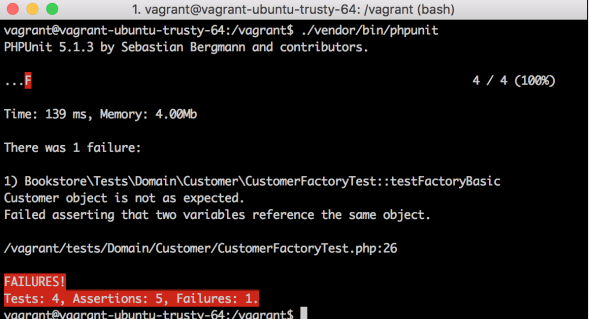
$customer,

$expectedBasicCustomer,

'Customer object is not as expected.'

);

A teszt eredménye a következő képernyőképen látható:



A teszt sikertelen volt, mert amikor két objektumot hasonlít össze az identitás összehasonlítással, az objektumreferenciát hasonlítja össze, és ez csak akkor lesz ugyanaz, ha a két objektum pontosan ugyanaz a példány. Ha két objektumot hozunk létre azonos tulajdonságokkal, akkor azok egyenlőek lesznek lesznek, de soha nem lesznek azonosak. A teszt javításához módosítsa az állítást a következőképpen:

$expectedBasicCustomer = new Basic(1, 'han', 'solo', 'han@solo.com');

$this->assertEquals(

$customer,

$expectedBasicCustomer,

'Customer object is not as expected.'

);

Most írjuk meg a teszteket az eladási domain objektumhoz a tests/Domain/SaleTest.php alatt. Ez az osztály nagyon könnyen tesztelhető, és lehetővé teszi számunkra néhány új állítás használatát, az alábbiak szerint:

<?php

namespace Bookstore\Tests\Domain\Customer;

use Bookstore\Domain\Sale;

use PHPUnit\_Framework\_TestCase;

class SaleTest extends PHPUnit\_Framework\_TestCase {

public function testNewSaleHasNoBooks() {

$sale = new Sale();

$this->assertEmpty(

$sale->getBooks(),

'When new, sale should have no books.'

);

}

public function testAddNewBook() {

$sale = new Sale();

$sale->addBook(123);

$this->assertCount(

1,

$sale->getBooks(),

'Number of books not valid.'

);

$this->assertArrayHasKey(

123,

$sale->getBooks(),

'Book id could not be found in array.'

);

$this->assertSame(

$sale->getBooks()[123],

1,

'When not specified, amount of books is 1.'

);

}

}

Itt két tesztet adtunk hozzá: az egyik biztosítja, hogy egy új eladási példány esetében a könyvek listája üres. Ehhez az assertEmpty metódust használtuk, ami egy tömböt fogad el argumentumként, és azt állítja, hogy az üres. A második teszt a egy könyv hozzáadása az eladáshoz, majd annak ellenőrzése, hogy a könyvek listája a megfelelő tartalma. Ehhez az assertCount metódust fogjuk használni, amely ellenőrzi, hogy a tömb, azaz a második argumentumnak annyi eleme van, mint a megadott első argumentumnak. Ebben az esetben azt várjuk, hogy a könyvek listája csak egy bejegyzést tartalmaz. A második állítás ennek a tesztnek annak az ellenőrzése, hogy a könyvek tömbje tartalmaz-e egy adott kulcsot, ami a könyv ID-je, az assertArrayHasKey metódussal, amelyben az első argumentum a kulcs, a második pedig a tömb. Végül a már ismert assertSame metódussal, hogy a beillesztett könyvek mennyisége 1.

Bár ez a két új állításmódszer néha hasznos lehet, mindhárom módszer utolsó teszt három állítása helyettesíthető egy assertSame metódussal, amely összehasonlítja a a könyvek teljes tömbjét az elvártal, az alábbiak szerint:

$this->assertSame(

[123 => 1],

$sale->getBooks(),

'Books array does not match.'

);

Az eladási tartomány objektum tesztcsomagja nem lenne elég, ha nem tesztelnénk, hogyan viselkedik az osztály több könyv hozzáadásakor. Ebben az esetben a assertCount és assertArrayHasKey használata szükségtelenül hosszúvá tenné a tesztet, ezért csak hasonlítsuk össze a tömböt egy elvárt tömbbel a következő kód segítségével:

public function testAddMultipleBooks() {

$sale = new Sale();

$sale->addBook(123, 4);

$sale->addBook(456, 2);

$sale->addBook(456, 8);

$this->assertSame(

[123 => 4, 456 => 10],

$sale->getBooks(),

'Books are not as expected.'

);

}

## Kivételek várása

Néha egy metódusnak bizonyos váratlan felhasználási esetekben kivételt kell dobnia. Amikor ez történik, megpróbálhatjuk a tesztben rögzíteni ezt a kivételt, vagy kihasználhatjuk a PHPUnit által kínált másik eszközt: a kivételek várását. Ahhoz, hogy egy tesztet úgy jelöljünk meg, hogy egy adott kivételt várjon, csak adjuk hozzá a @expectedException megjegyzést, amelyet a kivétel osztályának teljes neve követ. Opcionálisan használhatja a @expectedExceptionMessage megjegyzést a kivétel üzenetének megerősítéséhez. Adjuk hozzá a következő teszteket a CustomerFactoryTest osztályunkhoz:

/\*\*

\* @expectedException \InvalidArgumentException

\* @expectedExceptionMessage Wrong type.

\*/

public function testCreatingWrongTypeOfCustomer() {

$customer = CustomerFactory::factory(

'deluxe', 1, 'han', 'solo', 'han@solo.com'

);

}

Ebben a tesztben megpróbálunk létrehozni egy deluxe ügyfelet a gyárunkkal, de mivel ez a ügyféltípus nem létezik, egy kivételt fogunk kapni. A várt típus kivétel az InvalidArgumentException, a hibaüzenet pedig a következő: "Wrong type"(Rosz típus). Ha lefuttatjuk a teszteket, akkor látni fogjuk, hogy azok sikeresek.

Ha definiáltunk egy elvárt kivételt, és a kivétel soha nem kerül elő, akkor a teszt sikertelen lesz; a kivételek elvárása csak egy másik típusú állítás. Hogy ez megtörténjen, adjuk hozzá a következőt a tesztünkhöz, és futtassuk le; hibát fogunk kapni, és a PHPUnit panaszkodni fog, hogy várta a kivételt, de az nem került elő:

/\*\*

\* @expectedException \InvalidArgumentException

\*/

public function testCreatingCorrectCustomer() {

$customer = CustomerFactory::factory(

'basic', 1, 'han', 'solo', 'han@solo.com'

);

}

## Adatszolgáltatók

Ha végiggondoljuk a tesztek menetét, a legtöbbször egy metódust hívunk meg egy bemenettel, és várunk egy kimenetet. Annak érdekében, hogy az összes szélsőséges esetet lefedjük, természetes, hogy megismételjük ugyanazt a műveletet egy sor bemenettel és várható kimenettel. A PHPUnit lehetőséget ad erre, így rengeteg duplikált kódot távolíthatunk el. Ezt a funkciót nevezzük **adatszolgáltatásnak**.

Az adatszolgáltató a teszt osztályban definiált nyilvános metódus, amely egy tömböt ad vissza egy adott sémával. A tömb minden egyes bejegyzése egy olyan tesztet képvisel, amelyben a kulcs a a teszt neve - opcionálisan használhatunk numerikus kulcsokat is -, és az érték a paraméter, amelyre a tesztnek szüksége van. Egy teszt kijelenti, hogy szüksége van egy adatszolgáltatóra a következővel a @dataProvider annotációval, és a tesztek végrehajtásakor az adatszolgáltató behelyezi a az argumentumokat, amelyekre a tesztmódszernek szüksége van. Nézzünk egy példát, hogy könnyebbé tegyük. Írja a következő két metódust a CustomerFactoryTest osztályba:

public function providerFactoryValidCustomerTypes() {

return [

'Basic customer, lowercase' => [

'type' => 'basic',

'expectedType' => '\Bookstore\Domain\Customer\Basic'

],

'Basic customer, uppercase' => [

'type' => 'BASIC',

'expectedType' => '\Bookstore\Domain\Customer\Basic'

],

'Premium customer, lowercase' => [

'type' => 'premium',

'expectedType' => '\Bookstore\Domain\Customer\Premium'

],

'Premium customer, uppercase' => [

'type' => 'PREMIUM',

'expectedType' => '\Bookstore\Domain\Customer\Premium'

]

];

}

/\*\*

\* @dataProvider providerFactoryValidCustomerTypes

\* @param string $type

\* @param string $expectedType

\*/

public function testFactoryValidCustomerTypes(

string $type,

string $expectedType

) {

$customer = CustomerFactory::factory(

$type, 1, 'han', 'solo', 'han@solo.com'

);

$this->assertInstanceOf(

$expectedType,

$customer,

'Factory created the wrong type of customer.'

);

}

A teszt itt a testFactoryValidCustomerTypes, amely két argumentumot vár: $type és $expectedType. A teszt ezeket használja fel egy ügyfél létrehozására a gyárral és ellenőrzi az eredmény típusát, amit már megtettünk a típusok keménykódolásával. A teszt azt is deklarálja, hogy szüksége van a providerFactoryValidCustomerTypes adatszolgáltatóra. Ez az adatszolgáltató egy négy bejegyzésből álló tömböt ad vissza, ami azt jelenti, hogy a teszt négyszer fog végrehajtódni négy különböző argumentumkészlettel. A a neve minden egyes tesztnek az egyes bejegyzések kulcsa - például "Alapvető ügyfél, kisbetűs". Ez nagyon hasznos abban az esetben, ha a teszt sikertelen, mert a hiba részeként jelenik meg üzenet formájában. Minden bejegyzés egy leképezés két értékkel, a type és az expectedType értékekkel, amelyek a következők a tesztmódszer argumentumainak nevei. Ezeknek a bejegyzéseknek az értékei a értékek, amelyeket a tesztmódszer kapni fog.

A lényeg az, hogy a kód, amit írtunk, ugyanaz lenne, mintha a testFactoryValidCustomerTypes négyszer írtuk volna meg, a $type és $expectedType kemény kódolásával minden alkalommal. Képzeljük el most, hogy a tesztmódszer több tíz sornyi kódot tartalmaz, vagy azt szeretnénk, hogy ugyanazt a tesztet több tíz adatkészlettel megismételjük; látja, milyen erős ez?