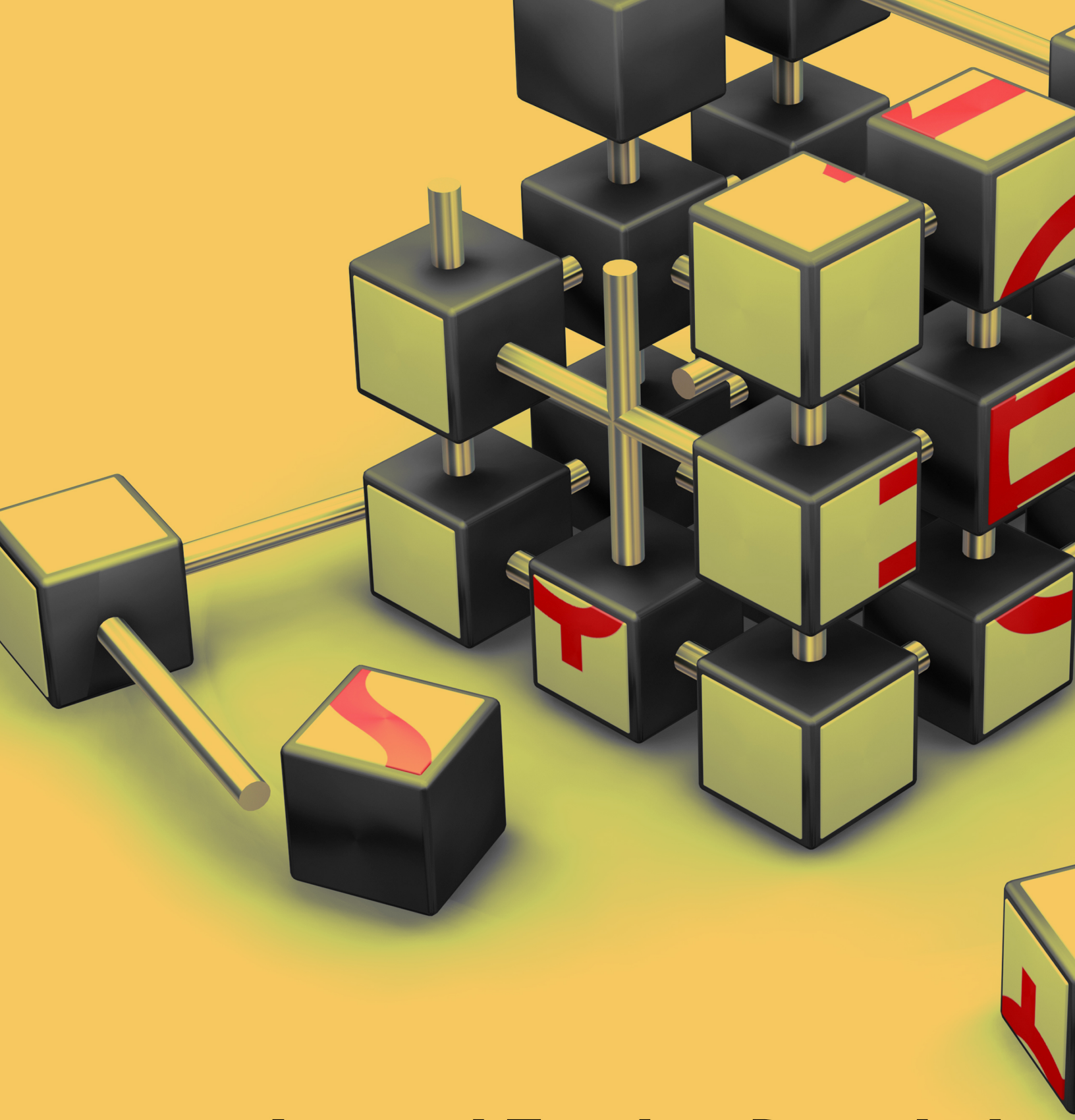


Laravel Testing Decoded

...with Jeffrey Way

Laravel Tesztelés Egyszerűen (Magyarul!)

A tesztelési könyv, amelyre mindig is vágytatok!

JeffreyWay and Zsellér István

This book is for sale at http://leanpub.com/laravel-testing-decoded-hungarian

This version was published on 2013-08-30

Tartalomjegyzék

Legyetek üdvözölve

Túl sokszor láttam ezt már. Ahogyan egy applikáció növekszik, ugyanúgy a ronda, teszteletlen kód is. Nemsokára úgy érzed magad, mint aki fulladozik, ahogyan megpróbálod kézi vezérlésen tesztelni minden funkcióját! Ilyenkor jut eszedbe, sőt bizonyossá válsz, hogy szükség van automatikus tesztelésre. Megkockáztatom, már olvastál TDD könyvet, de, ahogyan minden dolog az életben, tapasztalat nélkül nem megy, és az "aha pillanat" – várat magára.

A gond az, hogy a tesztelés eléggé trükkös foglalatosság. Könnyedén előfordulhat az is, ahogyan a kódodat írtad, hogy tesztelhetetlen! Amit tudatosítani kell, az, hogy a tesztelés nem csak arra garancia, hogy a kódod működik, de ezzel a mintával te is jobb fejlesztővé válhatsz. A ronda, tesztelhetetlen, és kifacsart kódtól örökre megszabadulsz. Higgy nekem: mihelyt megkérded magadtól "*hogyan tesztelhetném ezt a dolgot*" mielőtt még egy sort is írnál, vissza fogsz emlékezni a régi önmagadra, és nevetve gondolsz majd azokra az időkre, amikor még középkori módszerekkel eszkábáltál programokat.

Légy üdvözölve a modern szoftverfejlesztés időszakában.

A szoftvertesztelés alaptörvényei (és a TDD) nyelvtől függetlenek, amikor is a végrehajtáshoz érünk, milliárd technikával és eszközzel szembesülünk. Ez a köny legalább annyira bevezetés a TDD-be, mint egy mélyenszántó analízis, hogyan kell ezt a Laravelben végezni.

Megkezdődött

Amikor programozási nyelvekr?l beszélünk, mindenkinek megvan a saját véleménye. Amikor pedig éppen a PHP-ről diskurálunk, készülj fel egy kis szívatásra . Annak ellenére, hogy ez a nyelv jócskán megizmosodott az elmúlt öt évben , mindig lesznek olyanok, akik nem tudják a PHP komolyan venni.

A hitetlenek nem látják az 5.5-ös verziót, vagy az OOP-ot, vagy a modern keretrendszereket, mint a Laravel, esetleg sosem hallottak a Composerről, nem beszélve a test-first fejlesztésről. Nem bizony, ők még mindig a régi, csúf PHP 4 kódról beszélnek, vagy egy rosszul összedobott WordPress sablonról 2008-ból.

Szebb a PHP mint a Ruby? Nem. Az API-ja inkonzisztens időnként? Biztosan. A közösség vezető -e az innovációban és szoftvertervezésben ? Definitíve nem. A kérdés akkor, MIÉRT? Miért dominál a PHP - 80 százalékos résszel – amikor a versenyben lévő nyelvek mindegyike elegánsabb? Az okokat máshol kell keresnünk. Tegyük fel, a tény, hogy létrehozol egy fájlt, `echo` *hello world* tartalommal, és mindjárt látod is a kimenetet a böngészőben sokkal inkább felhasználóbarát, mint azt magunknak be merjük vallani. Lehetséges, hogy a flexibilitása előnyére válik, nem pedig hátrányára.

> Mióta kell a könnyű felhasználhatóságot lekicsinyelni?

Az igazság az, hogy a PHP *nem az új lány a grundon.* Nem különösképpen sexy. Nincs bétában. De, tudod mit? Megcsinálja azt, amit mondanak neki. Beszéljenek, amennyit csak akarnak a PHP 4-ról. Ameddig azzal foglalkoznak, mi felhasználjuk mindazt, amit a nyelv legújabb vívmányai, és a körülötte létező ökoszisztéma nyújt.

A PHP lesajnálásának vége. A PHP reneszánsza már elkezdődött. Modern objektum-orientált technikákat használunk, csomagokat osztunk meg a Composeren keresztül, verzió kontrollt alkalmazunk és continuous integrationt, modern keretrendszereket használunk, hiszünk a tesztelésben (*nemsokára te is fogsz*), üdvözöljük az újonnan jövőket (és nem csapjuk senki orrára az ajtót), miközben a legszélesebb mosolyunkat mutatjuk.

A legjobb rész pedig, Laravel felhasználóként, TE is részese vagy a mozgalmunknak ! Amikor először csatlakoztam a Laravel IRC csatornájára, nem kellett sokat várni, és már valaki köszöntött is "*Légy üdvözölve a családunkban.*" Semmi sem írja le szebben a közösségünket. Egyek vagyunk. Ez az a PHP közösség, amelyet én szeretek.

Ha megvetted ezt a könyvet, valószínűleg szükséged van egy kis segítségre a tesztelésnél. A Laravel szellemében, üdvözöllek a családban. Tanuljuk meg ezt együtt.

Nekem írták ezt a könyvet?

A gondokat a technikai könyvek írásakor az okozza, hogy hol húzzuk meg a határt, abból a szempontból, hogy az olvasóktól milyen előtudást követeljünk meg. Ha már hallottál a következő dolgokról, akkor vágj bele !

- PHP 5.3
- Laravel 3 (lehetőleg 4)
- Composer

Miért Laravel specifikus?

A könyvben bemutatott technikák nagy része akármilyen nyelvre,vagy keretrendszerre alkalmazható, de, tapasztalatom szerint , ebben az új világban az első lépéseket a legkényelmesebb cipőkben kell megtenni. Megtanulható a teszt-vezérelt fejlesztés egy Java könyvből? Fogadni mernék rá ! Könnyebb lenne-e egy nyelven, és keretrendszerrel, amelyet már ismerek? Minden bizonnyal.

Másodsorban, egy általános tesztelési könyvben nem tudnám demonstrálni a PHP-specifikus funkciók és csomagok nagy részét, amelyet személyesen használok mindennapi munkámban. Körbeöleli ez a PHPUnit segítőcsomagokat, az acceptance testing keretrendszerekig, mint a Codeception.

Remélem, ahogyan végigolvasod ezt a könyvet , tesztelési ismereteid bővítése mellett , sok Laravel-specifikus tippet és trükköt is megismersz.

Gyakorlatok

Itt-ott ebben az irományban, *Gyakorlatok-nak* nevezett részleteket fogsz látni. Gondolj ezekre, mint részletes HOGYAN-okra, amelyeket neked kell megvalósítanod, a fejezet írása közben. Az elmélet önmagában nem elég; a programok írása az, amely ezeket a mintákat, és technikákat megjegyezhetővé teszi.

Tehát, amikor egy *Gyakorlatok* fejezethez érsz, kapcsold be a számítógépet, és programozzunk együtt!

Hibák

Kérlek, tartsd észben azt, hogy megtettem mindent ami emberileg lehetséges, hogy ez a mű lehetőleg hibamentes legyen, de én is csak halandó vagyok. Ha valami hasonlót észlelsz, kérlek jelentsd a GitHubon[1], én pedig javítom, amint lehet. A legkitartóbb hibaírtók jutalma csoportos ölelés.

Hogyan olvasd ezt a könyvet

Ha kedved van az elejétől a végéig, de kezdheted azzal a fejezettel, amelyik legjobban érdekel. Minden fejezet egy egészet alkot. Például ha már ismered a unit testing alapjait, biztosan nem kell elolvasnod a “Unit Testing 101” fejezetet! Lapozz át rajta, én addig behunyom a szemem.

Jelentkezz nekem

Úgy tűnik sok időt fogunk együtt tölteni, közös munkálkodásunk során. Ha szeretnél közelebbről is megismerni, esetleg pár kérdésed lenne, ne felejts el beköszönni.

- **IRC (#laravel channel)**: JeffreyWay
- **Twitter**: @jeffrey_way[2]

[1] https://github.com/JeffreyWay/Laravel-Testing-Decoded
[2] http://twitter.com/jeffrey_way

Bele az ismeretlenbe

Az éj sötét fátylát borította ránk… Várjunk csak, ez egy másik történet. Szóval, az éj csöndes volt, csak a légkondi zörgött egy picit. A feleségem és az állataim már régen magamra hagytak, és az igazak álmát aludják. A kutyám, ahogyan általában, a legkitartóbb volt, de nem rovok semmit sem a terhére; ki vagyok én, hogy elítéljem őket azért, mert éjjel háromkor aludni szeretnének? Mindenesetre a körülmények éppen megfelelőek voltak. Éreztem legbelül. Ahogyan minden figyelmemet a laptopom képernyőjére irányítottam, a gondolataim egyszerre együvé forrtak .

A fejlesztők ismerik ezt az érzést: azok a ritka pillanatok, amikor, hirtelen, az azelőtt megfejthetetlen, most, legalábbis első pillantásra, értelmet nyert. Sokan ezt "*aha*" pillanatnak nevezik. Az első alkalom, amikor megértettem, mire jó a `<div>` olyan volt. Most már, persze, mindenkinek érthető, de ez nem tegnap volt. Minek pakolnám tele a HTML-t `<div>` -ekkel? A böngésző ugyanazt jeleníti meg! Az egyik nap azt mondták nekem, gondoljak rá mint kosarakra; tedd a HTML-t a kosárba, aztán, ha valamit odébb kell raknod, csak a `<div>` -et kell újra pozicionálni. És végre megértettem.

Egy csomó hasonló pillanatot idézhetnék fel, különösképpen a nehezen megjövő étvágyamat az objektum-orientált programozáshoz, interface-ek kódolására, végül teszt-vezérelt fejlesztésre.

Igen, az utóbbi dolog nem volt szerelem első látásra, be kell vallanom. Ahogyan a legtöbb fejlesztő is, olvastam egy könyvet, vagy egy cikket róla, aztán azt gondoltam, "Hmm, ez érdekesnek tűnik" aztán folytattam mindent ahogyan azelőtt. Végül is megmaradt a tudatomban, és az elején olyan lágy hangocska egyre hangosabb, és követelődzőbb lett.

> "*Ezt tesztelned kell, Jeffrey.*" "*Ha ezt már tesztelted volna, már nem kéne megint meg megint futtatnod a programot*" "*Ki fognak röhögni, ha erre a pull requestre nem írsz teszteket.*"

Mint ahogyan a legtöbb dolog az életben, az igazi változás erős elhatározást kíván, és nem árt, ha az egész világgal tudatjuk, "*Soha többé! Végeztem a régi dolgokkal.*" És megtettem. Ezrek tették ugyanezt. Most pedig rajtad a sor.

Ahogyan Leeroy Jenkins[3] mondaná, rendben: fogjunk neki! Ez az a tesztelési könyv, amelyre mindannyian vártatok.

[3] http://www.youtube.com/watch?v=LkCNJRfSZBU

Első fejezet: Tesztelj mindent

Minden létező teszteléssel foglalkozó könyv tartalmazza a "*Miért teszteljünk*" fejezetet. Ha egy kicsit dondolkozol rajta, , az, hogy megvetted ezt a könyvet már azt mutatja, hogy érdeklődsz a technika iránt. Azért nézzük csak meg miért tartják mások olyan fontosnak.

Az applikációk tesztelésének megtanulása, sajnos, kezdetben nem könnyű . Ez meglepőnek tűnhet; nekem legalábbis az volt! Az alapgondolat meglepően egyszerű: írj teszteket, hogy bebizonyítsd , hogy a kódod úgy működik, ahogyan elvártad.

> Állandóan a Google Chrome-ot nyitogatod, hogy valamely új funkciót tesztelj? Csukd csak be, helyette írj egy tesztet.

Hogyan tudna ez összezavarni? Nos, a dolgok gyorsan rázósak lehetnek, amikor utánanézel *mit* kell tesztelni.

> Kontrollereket kell tesztelnem? És a modellek? Foglalkozzam -e a nézetekkel is? Mi van a keretrendszer kódjával, az adatbázissal, vagy a web szervizek által kapott információval ? Mi van azzal a milliárd és egy fajta tesztelési technikával, amelyet a StackOverflow-n emlegetnek? A tesztelési keretrendszerek? PHPUnit? Rspec? Capybara? Codeception? Mink? A lista nagyon hosszú. Hol kell kezdenem?

Nem szeretek fogadni (*nos, bizonyos fokig, de leginkább a feleségemmel vívott Scrabble játékokra, amikor ország világ előtt zrikálhatom*[4] *ha nyerek*), de biztosra veszem, hogy a karriered során már felvetődtek ezek a gondolatok. A tesztelés egyszerű. Megérteni mit, és hogyan, egy egészen más történet. Reményeim szerint, ez a könyv segíteni fog ebben.

Már eddig is teszteltél

Az igazság az, hogy már régen a tesztelés mestere vagy . Ha már egyszer is leírtad, hogy `console.log`, vagy elküldtél egy űrlapot a web applikációdban, hogy lásd, hogyan működik valamilyen új funkció, akkor már teszteltél is. Még kölyökkorunkban is, mestertesztelők voltunk. "*Ha megcsavarom ezt a gombot , az ajtó kinyílik. Győzelem!*"

A gond az, hogy mindezen teszteket kézileg kellett végezni. Miért ügyködnél olyan dolgokon, amelyeket a számítógép el tud végezni helyetted(mellesleg sokkal gyorsabban)? A könyv célja, hogy a manuális teszteléstől eljussunk a teljesen automatikusig. Ez lehetővé teszi az állandó tesztelési kört, amelyeknél az applikáció fejlesztése közben egy-egy teszt hajtódik végre. Meglepően biztonságossá teszi ez a fejlesztést.

[4] http://notes.envato.com/team/jeffrey-wins-a-bet/

> "Amikor egy fejlesztő felfedezi a test-vezérelt fejlesztés tulajdonságait, olyan érzése támad, mintha egy új világ nyílna meg számára kevesebb stresszel, és bizonytalansággal."
> - DHH

6 nyerő tulajdonsága a TDD-nek

A tesztelés csalós dolog. Kezdetben azt hiszed, csak arra való, hogy meggyőződj, a kódod működik. Ez nem egészen igaz. A valóságban több előnye is van a TDD ciklusnak.

1. Biztonság

Ha hibázol, vagy egy létező funkciót teszel tönkre, a *test robot* azonnal jelenti. Képzeld el, hogy leírsz valamit, mented, majd abban a pillanatban üzenetet kapsz arról, hogy minden működik-e. Nem lenne nyugodalmasabb az álmod? Emlékszel arra a rondán írt osztályra, amelyet féltél megváltoztatni, mert túl sok külső kód függött tőle? Ha tesztekkel dolgozol, akkor nem kell tartani az ilyesmitől.

2. Részvétel

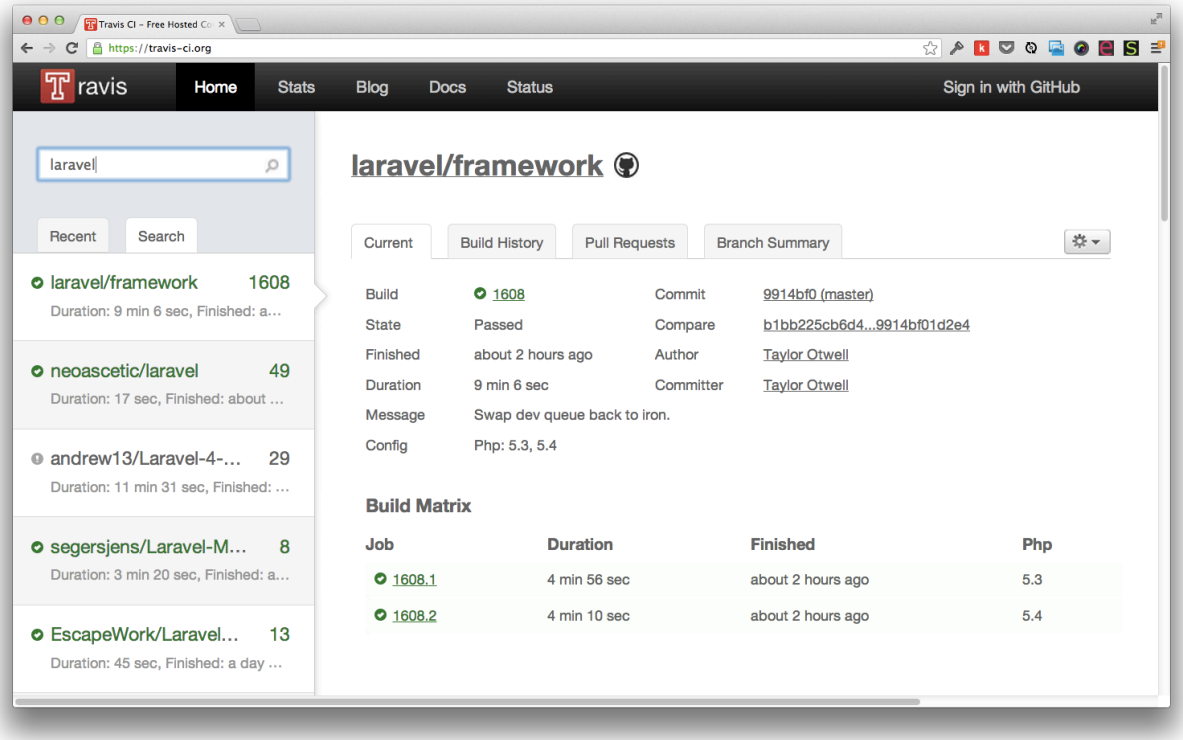
Ha nyílt forráskódú szoftveren szeretnél dolgozni, valószínűleg a közösségi fejlesztés vívmányait is fel szeretnéd használni a GitHubon . Egy idő után, mások is részt szeretnének venni, hogy javítsák a hibákat, vagy új funkcionalitást adjanak hozzá. Ha a projekted nem tartalmaz teszteket, és valaki pull requestet küld, hogyan ellenőrizhetnéd, ellenőrizhetnék, hogy nem történik semmi baj? A válasz? Nem - kivéve, ha minden felhasználási forgatókönyvet magad próbálsz ki. Kinek van erre ideje minden esetben?

A jól tesztelt projekt olyan, mint a kiválóan működő motor. Ha dolgozni szeretnél egy projekten, csak néhány lépést kell tenned:

1. A repo klónozása
2. Teszt írása, amely hibát generál (`testThrowsExceptionIfUserNameDoesNotExist`)
3. Változtasd meg a kódot, hogy a hiba eltűnjön
4. Futtasd a tesztet, ha a kimenet zölden villog akkor (*siker*)
5. Git commit, és pull request küldés

Van egy csomó continuous integration szerviz, mint a Travis[5], amely automatikusan futtatja a teszteket ha pull requestet kap . Ha a teszt nem sikerül, akkor a változás nem kerül elfogadásra.

[5] https://travis-ci.org/



travis-ci.org

3. Felnőttkorba lépés

Ha egy kicsit eltérünk a témától, viszont továbbra is a PHP közösségről beszélünk, a véleményem az, hogy a WordPress olyan, mint egy kétélű kard. Egyfelől lehetővé tette mindenkinek a bloggolást. Ez egy letagadhatatlan, tisztelendő tény. Könnyű rá sablonokat készíteni a profiknak. Készíts egy `index.php` fájlt, dobj be egy hurkot, amely előhívja a legújabb publikációkat, majd törődj a megjelenítéssel. Mi lenne egyszerűbb ennél?

Ez tulajdonképpen igaz. Ez mellett létrehozott egy PHP fejlesztői gárdát, amely csak a Wordpressben hajlandó részt venni. Ennek a következménye az lett, hogy modern minták és technikák , mint a teszt-vezérelt fejlesztés, MVC, verzió kontroll, nagyjából ismeretlenek számukra. Ez a kellemetlen igazság két mellékes eseménynek lett az alapja:

1. PHP közösség a legtöbb vitriolt a PHP 4 , és a WordPress kód miatt kapja.
2. A váltás a WordPresstől egy full-stack frameworkig, mint a Laravel, nagyon nehéz lehet. Az új minták, és eszközök használatának megtanulása nem könnyű.

A WordPress felelős ezért? Igen is, meg nem is. Annyi azonban bizonyos: nem nagyon törték magukat a mesterség fejlesztésén. A tesztelés 95%–ban ignorálva van (*ez persze csak egy kitalált szám*) az elérhető WordPress bővítményekben.

Végül is mindenkinek fel kell nőnie. Visszagondolhatunk a régi *kódolok, és közben nem gondolkodok* praktikákra. Ne tervezz, gondolkozz, ne tesztelj; bele a közepébe, *hadd szóljon*, közben pedig állandóan töltsd újra az oldalt, hátha valami sikerül.

Mi jobbak vagyunk ennél. **Mi fejlesztők vagyunk**. **Ne nézzenek pisztolyhősnek**.

Egy érdekes változás történik, amikor gondolkodsz is mielőtt kódolsz : a minőség azonnal javul. Ki gondolta volna? Gyorsan rájössz, hogy a tesztelés nem csak arra jó, hogy megbizonyosodj, hogy működik-e a kód. Amikor tesztelünk, használunk egy osztályt, vagy API-t még mielőtt megírták volna. Ez felszabadít, és növeli az olvashatóságot *Mi lenne a legjobban olvasható módszer, hogy adatot kapjunk egy web szerviztől?* Írd ezt le, nézd, ahogyan a teszt megbukik, majd tedd rendbe. Így működik!

4. A tesztelhetőség minőségi architektúrát biztosít

Amit megtanulhatsz a könyvet olvasva, hogy feltedd a kérdést , *Hogyan tesztelhetném ezt?* mielőtt egy sort is leírnál. Ez a kérdés lesz a biztonsági ernyőd, amely megvéd majd a múltbeli rossz tapasztalatoktól. Egyetlen metódus nem fog egy csomó funkcionalitást végezni, mert én lusta voltam rendesen megírni. Ez megnehezíti a tesztelést. A tesztelés javít a struktúrán is.

5. Dokumentáció

Hatalmas bónusz tesztírásnál, hogy dokumentációt biztosít. Szeretnéd tudni, hogy valamely osztály mit is tesz? Nézd át a teszteket, ha rendesen nevezték el őket (ami azt jelenti, hogy leírják a SUT-ot, vagy tesztelés alatt álló rendszert), semmi perc alatt érteni fogsz mindent!

6. Szórakoztató

Valljuk be : kockák vagyunk. Mely kocka nem szereti a jó játékot? Mellékhatás a teszt-vezérelt fejlesztésnek, hogy a munkát játékká alakítja. Hogyan színezem át a vörös kódot zöldre? Menj végig minden lépésen, ameddig eljutsz a célig. Először bután hangzik, de ígérem: szórakoztató lesz. Magad is megláthatod.

Mit kell tesztelnem?

Az alapvető törvény - amelyet sokszor meg fogok ismételni – tesztelj mindent ami tönkremehet. Lehetséges, hogy a definiált útvonalak közül egy *tönkremehet*, 404-es oldalt generálva? Igen? Akkor írj egy olyan tesztet, amely számol ezzel a lehetőséggel. Esetleg egy saját osztály, amely adatokat kér le egy táblából, majd ezeket egy fájlba írja ? Tesztelned kell azt is? Igen. Esetleg egy segédprogram,

amely a legújabb csiripeket ássa elő, és egy sidebarban jeleníti meg ?Ha az egyetlen alternatíva, hogy megnyisd a Google Chrome-ot, és megnézd, akkor a válasz IGEN.

A kellemetlen dolog, legalábbis az elején, hogy a *tesztelj mindent* mantra túl sok munkát jelent. Amúgy is nehéz megtanulni a tesztelés művészetét, ezért lassabban is lehet haladni. Kezd először a modellekkel. Ha megy, akkor próbálkozz a többi résszel. Nyugdíjasan. Megvan annak is miértje, miért tanulunk meg először az ujjainkon számolni.

Figyelmeztetés:

Ez a tipp csak félig igaz. Mindennek tesztelése gyönyörű cél, de *létezik* olyan kifejezés mint a túl-tesztelés (ezzel nem mindenki ért egyet). Egy növekvő része a közösségnek úgy tartja, hogy korlátozni kell a teszteket azokra a részletekre, ahol legszükségesebb. Tehát, ha ritkán hibázol, accessorok és mutatorok írásánál, ne kínlódj a tesztelésükkel. A teszteknek kell tégedet szolgálniuk, nem pedig fordítva.

A te feladatod, hogy megtanuld meghúzni a határt . Meg kell találni azt a pontot, amikor a befektetett munka már nem térül meg[a]?

[a] http://en.wikipedia.org/wiki/Diminishing_returns

6 A tesztelhetetlen kód jelei

A tesztelés kicsit olyan, mint utazni egy olyan országban, ahol senki sem beszéli a nyelvedet. Ahogyan felfedezed a vidéket, mintákat veszel majd észre. Rövid időn belül pedig folyékonyan kommunikálsz. Nem atomfizika, amelyről beszélgetünk itt, mindenki megtanulhatja, aki akarja. Csak nyomást…és időt igényel (*Olvasd ezt a mondatot Morgan Freeman hangján*).

Ahogyan jobb és jobb leszel, könnyebben észreveszed a macerás részeket. Ösztönösen észre fogod venni a hibás mintákat.

A legkönnyebb öt dolog, amikre oda kell figyelni:

1. New Operátorok

A tesztelés alaptörvénye, hogy mindent izolálva kell megfigyelni. Ezt az elkövetkező fejezetekben még mélyebben is megtárgyaljuk, de röviden az aktuális osztályt kell tesztelni, semmi mást. Ne kapcsolódj az adatbázishoz, ne teszteld, hogy a `Filesystem` osztály adatokat feccöl valami web szerviztől. Azoknak is kijár a saját tesztjük, ne keverjük őket.

Mihelyt telerakod `new` operátorokkal az oszályaidat, azonnal megszeged ezt a törvényt. Emlékeztetlek: az osztály izolált tesztelése röviden azt jelenti, hogy más objektumokat nem hozunk létre a kódrészletben.

Hibás minta:

```
public function fetch($url)
{
    // We can't test this!
    $file = new Filesystem;

    return $this->data = $file->get($url);
}
```

Ez egy olyan eset, amikor a PHP nem olyan flexibilis, amennyire szeretnénk. Míg Ruby-ban újranyithatjuk az osztályt (ismeretes mint monkey-patching) és felülírhatjuk a metódusokat (*amely különösen hasznos a tesztelésnél*), PHP, sajnos nem engedi meg - *legalábbis különleges bővítmények fordítása nélkül*. Ezért, használnunk kell egy technikát, amelyet dependency injectionnek hívnak.

Így jobb:

```
protected $file;

public function __construct(Filesystem $file)
{
    $this->file = $file;
}

public function fetch($url)
{
    return $this->data = $this->file->get($url);
}
```

Ezzel a változtatással , egy `Filesystem` osztályt utánzunk, tesztelhetővé téve a kódot. Ne zavartasd magad, ha az alábbi kódot nem érted. Nemsokára megtanulod majd a belső működését is! Egyenlőre olvasd át.

```
public function testFetchesData()
{
    $file = Mockery::mock('Filesystem');
    $file->shouldReceive('get')->once()->andReturn('foo');

    $someClass = new SomeClass($file);
    $data = $someClass->fetch('http://example.com');

    $this->assertEquals('foo', $data);
}
```

Az egyetlen eset, amikor egy osztályból objektumot hozunk létre egy másik oszályban, ha az objektum egy value-object, vagyis egy konténer getterekkel, és setterekkel, és semmilyen igazi munkát nem végez.

Tipp: Nem kell levadászni az összes osztályt , amely a `new` kulcsszavat tartalmazza.

2. Ellenőrzésmániás konstruktorok

A konstruktor egyetlen dolga, hogy a függőségeket meghatározza. Ezt úgy kell elképzelni, mintha az oszály azt *kérné Hozzájuthatnék a* `Filesystem` *osztályhoz?*

Ha ettől több történik, gondolkodj a változtatáson.

Hibás minta:

```
1  public function __construct(Filesystem $file, Cache $cache)
2  {
3      $this->file = $file;
4      $this->cache = $cache;
5
6      $data = $this->file->get('http://example.com');
7      $this->write($data);
8  }
```

Javítva:

```
1  public function __construct(Filesystem $file, Cache $cache)
2  {
3      $this->file = $file;
4      $this->cache = $cache;
5  }
```

A tesztelésnél a következő három dolgot tesszük :

1. Rendezünk
2. Végrehajtunk
3. Érvényesítünk

Ha egy osztály konstruktora tele van saját metódusokkal, minden egyes tesztedben számítanod kell rájuk.

Tipp: Csak egyszerűen : a konstruktorokban csak a függőségeket határozzuk meg.

3. És...

Előrelátni, hogy milyen feladatokkal bízzuk meg az osztályunkat nehéz lehet elsőre. Hallottuk eleget az *Egyetlen Feladat Törvényét*, de nehéz ezt a gyakorlatban megvalósítani.

4 mód, hogy észrevegyük a túl sok mindennel foglalkozó osztályokat

1. Egyszerűen csak sorold fel, mit kell egy osztálynak tennie. Ha túl sokszor mondod azt, hogy *és*, valószínűleg újra kell írni.
2. Tanuld meg azonnal analizálni a metódus összes sorát. Jó esetben a metódus csak párat tartalmaz (jó lenne, ha egyet). Ha túl sok sor van, akkor valószínűleg nincs minden rendben.
3. Ha nem tudsz jó nevet találni az osztályodnak, lehet, hogy a logika sántít.
4. Ha minden más bukik, mutasd meg valaki másnak. Ha azonnal nem értik meg mire jó az osztály (*Ez itt a jelszót hasheli *), változtasd meg.

Néhány példa kezdetnek:

- A `FileLogger` osztály adatok naplózásával foglalkozik.
- A `TwitterStream` osztály csiripeket tölt le a Twitter API-val , ha felhasználónevet adunk meg neki.
- A `Validator` osztály adatokat ellenőriz előre meghatározott szabályok szerint .
- A `SQLBuilder` SQL lekérdezést készít valamilyen adathalmazból.
- A `UserAuthenticator` osztály ellenőrzi a felhasználó beléptetését.

Figyeld meg a fenti leírásban egyetlen *és* sem tűnik fel. Így sokkal könnyebb tesztelni, mert nem kell több objektummal törődni.

Tipp: Egy osztály-egy feladat. Ez az *Egyetlen Feladat Törvénye.*

4. Túl sok útvonal? Polimorfizmust neki!

A polimorfizmus megértéséhez szükség van egy aha pillanatra[6]. Ha egyszer megérted, sosem feleded el.

Definíció: A polimorfizmus kifejezés egy komplex osztály felosztását jelenti sok hasonlóra, amelyek közös interfészt használnak, de saját funkcióval rendelkeznek.

[6] https://tutsplus.com/2012/04/the-aha-moment/

Ha rá akarunk jönni, vajon egy osztálynak jól jönne a polimorfizmus, akkor keresni kell a `switch` állításokat (vagy a túl sok feltételt). Képzelj el egy bankszámla osztályt, amelynek ki kell számolnia az évi kamatot a számlatípustól függően. Itt egy egyszerűsített példa:

```
function addYearlyInterest($balance)
{
    switch ($this->accountType) {
        case 'checking':
            $rate = $this->getCheckingInterestRate();
            break;

        case 'savings':
            $rate = $this->getSavingsInterestRate();
            break;

        // other types of accounts here
    }

    return $balance + ($balance * $rate);
}
```

Hasonló esetben értelmesebb megoldás lenne ha a hasonló logikát kisebb osztályok használnák fel.

Definiálj egy interfészt, hogy biztosan legyen egy `getRate` metódusod. Az interfészeket sokszor szerződésnek írják le. Minden implementáció a szerződésnek megfelelően működik.

```
interface BankInterestInterface {
    public function getRate();
}
```

Create a checking implementation of the interface.

```
class CheckingInterest implements BankInterestInterface {
    public function getRate()
    {
        return .01;
    }
}
```

Create a savings implementation of the interface.

```
class SavingsInterest implements BankInterestInterface {
    public function getRate()
    {
        return .03;
    }
}
```

Az eredeti metódus így sokkal érthetőbb. Az `$interest` változó típusára előre utalunk . Ezzel lehetetlenné válik a `getRate` metódus hívása, ha eredetileg nem definiáltuk. Ezért írtunk egy interfészt. Minden implementációja az interfésznek tartalmaz egy `getRate` metódust.

```
function addYearlyInterest($balance, BankInterestInterface $interest)
{
    $rate = $interest->getRate();

    return $balance + ($balance * $rate);
}

$bank = new BankAccount;
$bank->addYearlyInterest(100, new CheckingInterest); // 101
$bank->addYearlyInterest(100, new SavingsInterest); // 103
```

Még jobb az, hogy így könnyebb tesztelni, hiszen a funkció csak egyetlen úton haladhat. Ne idegeskedj a szintaxis miatt . Nemsokára azt is átvesszük.

```
public function testAddYearlyInterest()
{
    $interest = Mockery::mock('BankInterestInterface');
    $interest->shouldReceive('getRate')->once()->andReturn(.03);

    $bank = new BankAccount;
    $newBalance = $bank->addYearlyInterest(100, $interest);

    $this->assertEquals(103, $newBalance);
}
```

Tipp: A polimorfizmussal nagyobb osztályokat kisebbekre lehet felosztani, amelyeket gyakran *alosztályoknak* neveznek. Emlékeztetlek: minél kisebb az osztály , annnál könyebb tesztelni.

5. Túl sok függőség

Az *Ellenérzésmániás konstruktor * tippben , beszéltünk arról, hogy a függőségeket a konstruktoron keresztül kell meghatározni. Ha olyasmit tapasztalsz, hogy egy osztály túl sokat kér, valószínűleg nincs minden rendben vele.

> "Mindig újragondolom egy osztály felépítését, ha több mint négy [függőséget tartalmaz]." - *Taylor Otwell*[7]

Alapvető törvényszerűség az objektum-orientált programozásban, hogy összefüggés van az osztály, vagy metódus paraméterei számának, illetve flexibilitása között (tesztelhetőség!). Minden esetben, amikor megszabadulsz egy paramétertől, vagy függőségtől, javítsz a kódon.

Ha az osztályod túl sok függőséget tartalmaz, írd újra.

6. Túl sok hiba

Egyszer hallottam Ben Orensteintől azt, hogy a *hibák imádják a társaságot.* Ez eddig a legigazabb állítás, amit eddig hallottam. Ha túl sokkal találkozol egyetlen osztályon belül, akkor ez egy segélykiáltás alosztályok és újraírás után. Gondoljunk csak bele: a hiba létrejöttének egyik alapvető oka az, hogy első alkalommal sem értetted a kód által nyújtott megoldást; túl összetett volt ! Tudod mit? Összetett kód sokszor egyenlő a tesztelhetetlen kóddal. Ez aztán egy csomó hibához vezet, és, mint tudjuk, a baj nem jár egyedül.

Definíció: Coupling -csatolás- szó azt jelüli, hogy a rendszerben két komponens mennyire függ egymástól. Ha az egyik hiánya hatással van a másikra, akkor tightly coupled coderól -erősen csatolt- beszélünk.

Ahogyan Ben mondta, ha hiba volt a hetedik sorban, akkor jó esély van rá, hogy gond lesz a tizenegyedikben is. Vésd ezt a fejedbe.

Tipp: Ha hasonló hibákkal találkozols, gondolkodj el arról, hogyan lehetne kisebb (tesztelhető) osztályokba pakolni a kódot. A kód nem csak tesztelhetőbb lesz, de könnyebben lehet majd olvasni.

A baj nem jár egyedül. Kár már az elsőt is keresgélni.

[7] https://twitter.com/taylorotwell/status/334789979920285697

Teszt zsargon

Annyira nem vagyok elszálva magamtól, hogy ebből a könyvből fogsz mindent megtanulni a tesztelésről (Legalább is reménykedem). Ha hasonlítunk, akkor késő éjjel is a webet fogod böngészni infók után.

E közben mindenféle zavaró kifejezéssel találkozhatsz. Ettől már csak az a rosszabb, hogy a terminológia minden nyelvben más! Anyám! Ebben a könyben – az egyszerűség szellemében – mindent a lehető legegyszerűbben próbálok magyarázni. Nézd át a következő definíciókat, de most nem muszáj megjegyezni őket. Az igazság az, hogy nem szeretem a sok zsargont. Ha a szó elsőre nem jelent semmit, akkor meg kell változtatni. A fejlesztőknek is lehet valamit tanulni az asztrofizikusoktól.

> "A legérthetőbb tudomány a nyelv szempontjából, szerintem, az asztrofizika. What do you call spots on the sun? Hogyan hívod a foltokat a Napon? Napfoltoknak. Azon részeit az űrnek, ahova belezuhansz, és sosem jössz ki? Fekete lyukaknak. Nagy vörös csillagokat? Vörös óriásnak. Adok egy feladatot kollegáimnak. Ő mond egy szót, ha pedig megértettem, azt mondom, "Ó, ez azt jelenti, hogy da-da-da-de-da?" - Neil Degrasse Tyson

Egységtesztelés

Gondolj az egységtesztelésre, mintha egy finom fésűvel mennél át minden osztályon, és metóduson, hogy megbizonyosodj, minden megfelelően működik. Az egységtesztelést izolációban kell végezni, hogy a hibakeresés a lehető legegyszerűbb legyen. 80% tesztjeidnek ilyen lesz.

Ha ez segít, ha az egységtesztelésre gondolsz, mondogasd *egy objektum, és csak egy objektum.* Ha a teszt nem sikerül, azonnal tudni fogod hol a hiba.

Modell Tesztelés

Bizonyos Ruby on Rails közösségi tagok modell tesztelésre gondolnak (még akkor is ha a tesztek adatbázisforgalommal járnak) egységteszteléskor. Ez egy kicsit csalós lehet. Az egységteszt izoláltan működik a külső függőségektől. Ha ezt az alapvető szabályt nem tartjuk be , akkor már nem egységtesztet hajtunk végre. Ez ekkor már integrációs teszt (erről nemsokára többet).

Ebben a könyvben, ha a modelleket teszteljük, ehhez a törvényszerűséghez tartjuk magunkat .

Képzeld el, hogy egy metódus a modelledben email küldésre való. Ha a megfelelő mintát követed, akkor van egy csak az emailek küldésével foglalkozó osztályod (*Egyetlen Felelősség Törvénye*). Ebben az esetben viszont egy gonddal szembesülünk: hogyan tesztelhetjük ezt a metódust, ha egy külső `Mailer` osztályt hív? Ekkor utánzásokat használunk, amelyekről még sokat hallasz ebben a könyvben. Egy hamis `Mailer` osztályt hozunk létre, és azt várjuk, hogy a megfelelő metódust hívják. Így, még ha a `Mailer` komponens ebben a pillanatban nem működik (*saját tesztjei lesznek*), megbizonyosodhatunk arról, hogy a modell úgy működik, ahogyan elképzeltük.

Integráció Tesztelés

Ha az egységteszt bizonyította, hogy a komponens működik izolációban, az integrációs teszt éppen az ellenkezője. Ezek a tesztek sok részét próbálják ki az applikációnak, és alapvetően nem használnak utánzatokat. Ilyenkor készíts egy erre a célra kijelölt adatbázist.

Például egy autó. Mondjuk a motor és a tank külön-külön működik (átmentek az egységteszten), de tudnak-e együtt dolgozni? Az integráció tesztelés erre való.

Funkcionális (Kontroller) Tesztelés

Hogy hívjuk a kontrollerek tesztelését? Bizonyos keretrendszerekben ezt funkcionális teszteknek nevezzük (végül is azok), de mi -dobpergés - *kontroller tesztelésnek* hívjuk!

Tradicionálisan a funkció tesztelés arra jó, hogy te, és a társaid megbizonyosodjatok arról, hogy a kód azt teszi, amit elvárnak tőle. Az egységtesztelés mindig egy *egységén* az osztálynak működik, a funkcionális több részét használhatja az applikációnak. Ezek a tesztek leggyakrabban kívülről jönnek, ezért is hívják *Rendszer Teszteknek*. Egy fontos törvény, hogy a funkcionális tesztelés nem igényel szervert.

Elfogadási teszt

Már megtanultuk, hogy a funkcionális tesztek arra jók, hogy a fejlesztői gárda elvárásai szerint próbáljuk ki a programot. Ennek ellenére, előfordulhat, hogy a teszt sikeres, viszont a kliens nincs megelégedve az eredménnyel. Ilyenkor van szerepe az elfogadási tesztnek. Másképpen, ez a kód megfelel -e a kliens elvárásainak? A program átmehet mindenféle egység-, funkcionális, integrációs teszten , de megbukik az elfogadásin, mert a kliens rájött, hogy nem minden úgy van, ahogyan elvárta.

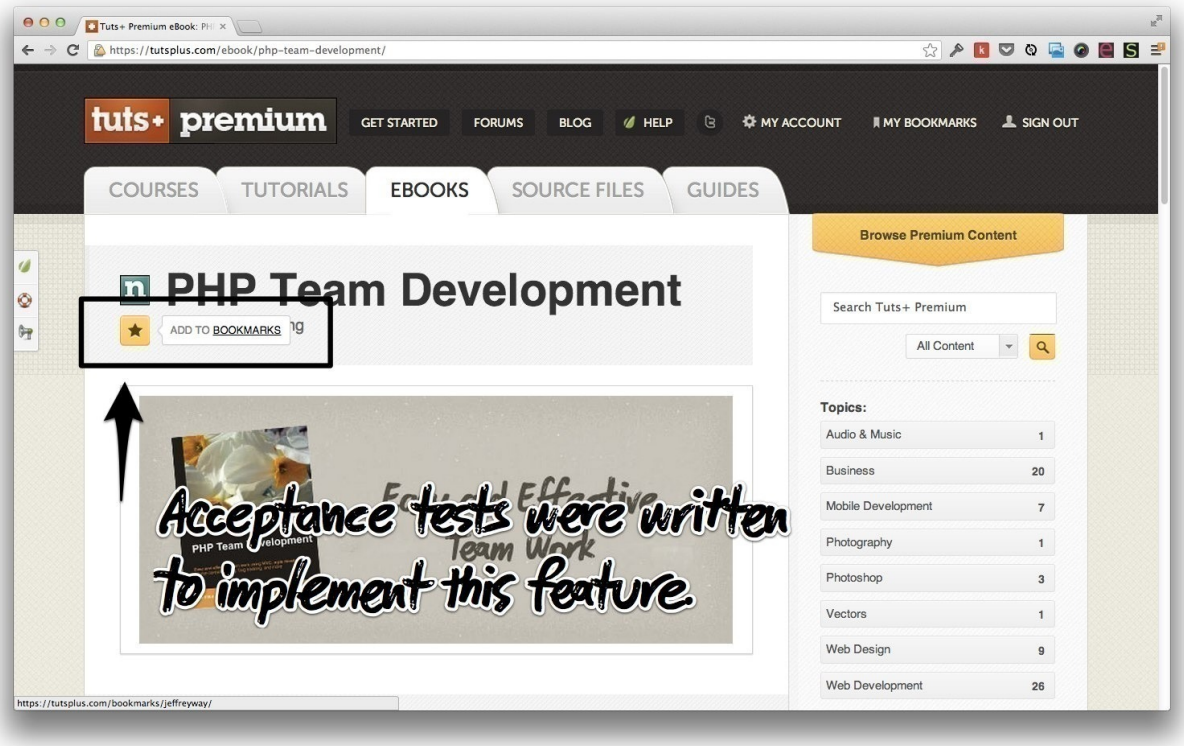
Tipp: A funkcionális tesztek a fejlesztők elvárásait igazolják, az elfogadásiak pedig a kliensét.

Gondolj a Tuts+ Premiumra[8], egy edukációs szolgáltatásra, amelyen dolgozom. Nemrégiben , hozzáadtunk egy funkciót, amely lehetővé teszi könyvjelzők létrehozását, amelyekkel elmenthetik a tanfolyamokat, és könyveket, amelyeket később szeretnének elolvasni . Mielőtt a fejlesztők egyetlen sor kódot leírnának, meg kell érteniük, hogy mit is várnak el tőlük a tartalmakért felelős csapat tagjai – hiszen ők kérték a funkciót. Ez egy elfogadási tesztet igényel.

[8] http://tutsplus.com

```
1 Meg szeretném jelölni azt amit tanulni akarok
2 Bejelentkezett felhasználó vagyok
3 Könyvjelzőt akarok létrehozni
```

Ha ez az elfogadási teszt átmegy, akkor az egész funkció teljességében elfogadott, megfelel a kliens (ebben az esetben a tartalmakért felelős csapat) elvárásainak.



A könyvjelző funkció a Tuts+ Premiumon elfogadási teszteket igényelt.

A könyv végén megtanulod, hogyan kell elfogadási teszteket írni a Codeception keretrendszerrel. Ezzel *emberi nyelven* definiálhatjuk, hogyan kell működnie az applikációnak.

A tesztelés költséges?

Elterjedt nézet az, hogy a tesztek ugyan jól jönnek, de amikor igazi munkáról van szó, a kliens pénztárcája szabja meg a határt.E szerint nincs az a kliens, aki megduplázná a fizetett összeget a nagyobb biztonságért.

Van ennek alapja? Nem. Sok tanulmány szól arról, hogy a teszt-vezérelt fejlesztés megrövidíti az előállítási ciklust .

Nyugi

Be kell vallanom, hogy ezek a szavak, és technikák sokáig idegenek voltak számomra is. Nem várom el, hogy mindent értsetek, és emlékezzetek rájuk. Nyugi; még el sem jutottunk a "*Bevezetés a PHPUnitba*" fejezetig! Egyenlőre meg kell jegyezni, hogy ameddig egy applikációt fejlesztesz sokféle tesztelési módszert fogsz használni.

A kellemetlen igazság az, hogy a fejlesztői közösség csak nem tud megegyezni a terminológián. Látni fogsz kifejezéseket, mint rendszer teszt, kérési specifikációk, medium tesztek, és még sok más. Legtöbbször, átfedések vannak a fentebb említett kifejezések között. Ne zavarjon ez most téged; a legfontosabb, hogy elkezdj tesztelni. Ki fogod alakítani a saját stílusodat.

> Ahogyan mondják, mindegy hogyan tesztelsz... ameddig tesztelsz.

Nem csak a terminológiával van így . Mostanság a legtöbb fejlesztő elfogadja, hogy nagyon fontos teszteket írni, viszont, ahogyan írják őket, már nagyon különbözik. Bizonyos személyek, mint Bob Martin (Uncle Bob), TDD filozófia szigorú betartását ajánlja : egyetlen sor produkciós kódot se írj, ameddig nem írtál tesztet rá.

> "Lehetetlenné vált egy szoftverfejlesztő számára, hogy profinak nevezze magát, ha nem végez teszt-vezérelt fejlesztést. " - Bob Martin[9]

Némely, szintén prominens programozók, mint DHH (Ruby on Rails létrehozója), nyugodtan bevallja, hogy ők a teszteket a produkciós kód után írják- úgy 80 százalékban.

> "Ne törd magad, hogy előzetesen tesztelj minden modellt, kontrollert, és nézetet (az én egyenlegem 20% előzetes, 80% utótesztelés)." - David Heinemeier Hansson[10]

A te munkád az, hogy a lehető legtöbb helyről szerezz információt, majd alakítsd ki azt a stílust, amit te (vagy a fejlesztői csapat) kedvel. Ez a könyv nem Szentírás, sokkal inkább sajátos adaptációja a tesztelésnek, amelyet magadévá tehetsz.

[9] http://www.youtube.com/watch?feature=player_detailpage&v=KtHQGs3zFAM#t=77s
[10] http://37signals.com/svn/posts/3159-testing-like-the-tsa