From Apprentice To Artisan

Laravel 4 İle İleri Düzey Uygulama Mimarisi

Yazarlar Taylor Otwell & Sinan Eldem

Laravel: From Apprentice To Artisan (TR) Türkçe

Laravel 4 İle İleri Düzey Uygulama Mimarisi

Taylor Otwell ve Sinan Eldem

Bu kitap http://leanpub.com/laravel-4-tr adresinde satıştadır.

Bu versiyon, 2015-08-29 tarihinde yayınlanmıştır

Bu Kitabı Tweet'le!

Yazara, Taylor Otwell ve Sinan Eldem, destek olmak için bu kitabı Twitter 'da paylaşın!

Bu kitap için önerilen hashtag #laravel-4-tr.

Bu linke tıklayarak, Twitter'da bu kitap hakkında neler paylaşıldığını görebilirsiniz:

https://twitter.com/search?q =#laravel-4-tr

İçindekiler

Bağımlılık Enjeksiyonu

Problem

Laravel framework'ün temeli, onun güçlü IoC konteyneridir. Frameworkün gerçek anlamda anlaşılabilmesi için, konteynerin güçlü bir şekilde kavranması gerekmektedir. Bununla birlikte, bir IoC konteynerinin sadece bir yazılım tasarım deseni (*dependency injection*) gerçekleştirmek için kolaylık sağlayan bir mekanizma olduğunu belirtmeliyiz. Bağımlılık enjeksiyonunu gerçekleştirmek için bir konteyner zorunlu değildir, sadece bu görevi daha kolay hale getirir.

İlk olarak, bağımlılık enjeksiyonunun neden faydalı olduğunu inceleyelim. Aşağıdaki sınıfı ve metodu ele alalım:

```
1  class UserController extends BaseController {
2
3          public function getIndex()
4          {
5                  $users = User::all();
6
7                  return View::make('users.index', compact('users'));
8          }
9
10 }
```

Bu kod özlü olsa da, gerçek bir veritabanı olmaksızın onu test edemeyiz. Diğer bir deyişle Eloquent ORM, controller'imizle *sıkıca bağlanmış* tır. Ayrıca, canlı bir veritabanına ulaşım da dahil olmak üzere bütün Eloquent ORM'yi kullanmadan da bu controller'i test etme imkanımız yoktur. Bu kod çoğunlukla *separation of concerns* (ilgilerin ayrılığı) adı verilen bir yazılım tasarım ilkesini de ihlal etmektedir. Basitçe söylemek gerekirse: controller'imiz çok şey biliyor. Controller'lerin verilerin *nereden* geldiğini bilmesine gerek yoktur, sadece ona nasıl ulaşacağını bilmesi gerekir. Bir controller'in verinin MySQL'de olduğunu bilmesine gerek yoktur, sadece onun *bir yerlerde* bulunuyor olduğunu bilmesi gerekir.

Separation Of Concerns (İlgilerin Ayrılığı)

Her sınıfın tek bir sorumluluğu olmalıdır ve bu sorumluluk sınıf tarafından tam olarak enkapsüle edilmiş olmalıdır.

Bu yüzden, web katmanımızın (controller) veri erişim katmanımızdan tamamen ayrı tutulması bizim için yararlı olacaktır. Bu bizim depolama uygulamalarımızda kolayca geçişler yapabilmemizi, bunun yanında kodun test edilebilirliğini de kolaylaştırmamızı sağlayacaktır. "Web"i sadece "gerçek" uygulamanıza bir ulaştırma katmanı olarak düşünün.

Uygulamanızı çeşitli kablo girişleri olan bir monitör olarak hayal edin. Monitörün işlevselliğine HDMI, VGA veya DVI aracılığı ile erişebilirsiniz. İnterneti de uygulamanıza giren bir kablo olarak düşünün. Bir monitörün işlevselliğinin hacmi kablodan bağımsızdır. Kablo tıpkı HTTP'nin sizin uygulamanız için bir ulaştırma katmanı olması gibi sadece bir ulaştırma mekanizmasıdır. Bu yüzden, ulaştırma mekanizmamızı (controller) uygulama mantığımızla kaplamak istemeyiz. Bu, bir API ya da mobil uygulama gibi herhangi bir ulaştırma katmanının uygulama mantığımıza ulaşmasına imkan verecektir.

Bu yüzden, controller'imizi Eloquent ORM'ye kaplatmak yerine, bir repository (ambar) sınıfı enjekte edelim.

Bir Sözleşme İnşa Edin

Öncelikle bir interface ve ona karşılık gelen bir implementation tanımlayacağız:

```
1  interface UserRepositoryInterface {
2
3          public function all();
4
5  }
6
7  class DbUserRepository implements UserRepositoryInterface {
8
9          public function all()
10         {
11                 return User::all()->toArray();
12         }
13
14 }
```

Sonra da controller'imize bu interface'in bir implementation'unu enjekte edeceğiz:

```
class UserController extends BaseController {

        public function __construct(UserRepositoryInterface $users)
        {
                $this->users = $users;
        }

        public function getIndex()
        {
                $users = $this->users->all();

                return View::make('users.index', compact('users'));
        }

}
```

Artık controller'imiz user verisinin nerede saklandığı konusunda tamamen bilgisizdir. Böyle bir durumda, cehalet mutluluktur! Verilerimiz MySQL, MongoDB veya Redis'ten geliyor olabilir. Controller'imiz bu farkı bilmediği gibi önemsemez de. Sadece bu küçük değişikliği yapmakla, web katmanımızı veri katmanımızdan bağımsız olarak test edebildiğimiz gibi, depolama implementasyonlarımızı da kolayca değiştirebiliriz.

Saygı Sınırları

Sorumluluk sınırlarına saygı göstermeyi unutmayın. Controllerler ve rotalar HTTP ile uygulamanız arasında bir aracı olarak hizmet ederler. Büyük uygulamalar yazarken, bunları domain mantığınızla kaplamayın.

Anlaşılmasını kuvvetlendirmek için en iyisi hızlı bir test yazalım. İlk olarak repository'yi mock (taklit) edeceğiz ve onu uygulama IoC konteynerine bağlayacağız. Sonra da, controllerin bu repository'yi düzgün bir biçimde çağırdığından emin olacağız:

```php
public function testIndexActionBindsUsersFromRepository()
{
        // Arrange...
        $repository = Mockery::mock('UserRepositoryInterface');
        $repository->shouldReceive('all')->once()->andReturn(array('foo'));
        App::instance('UserRepositoryInterface', $repository);

        // Act...
        $response = $this->action('GET', 'UserController@getIndex');

        // Assert...
        $this->assertResponseOk();
        $this->assertViewHas('users', array('foo'));
}
```

Beni Taklit Ediyor Musunuz?

Bu örnekte, `Mockery` taklit etme kitaplığını kullandık. Bu kitaplık sınıflarınızı taklit etmede temiz, etkileyici bir interface sağlar. Mockery, Composer aracılığıyla kolaylıkla yüklenebilir.

Daha İlerisi

Öğrendiklerimizi daha da kuvvetlendirmek için başka bir örneği ele alalım. Belki müşterilerimizi hesaplarına yapılan ücretlendirmeler konusunda bilgilendirmek istiyoruz. İki tane interface veya sözleşme tanımlayacağız. Bu sözleşmeler bize bunların implementation'larını daha sonra değiştirebilme esnekliği verecektir.

```php
interface BillerInterface {
        public function bill(array $user, $amount);
}

interface BillingNotifierInterface {
        public function notify(array $user, $amount);
}
```

Sonra da, `BillerInterface` sözleşmemizin bir implementasyonunu inşa edelim:

```
class StripeBiller implements BillerInterface {

        public function __construct(BillingNotifierInterface $notifier)
        {
                $this->notifier = $notifier;
        }

        public function bill(array $user, $amount)
        {
                // Stripe aracılıyla faturala...

                $this->notifier->notify($user, $amount);
        }

}
```

Her sınıfın sorumluluklarını ayırmak suretiyle, şimdi faturalama sınıfımıza kolaylıkla çeşitli bilgilendirme implementasyonları enjekte edebileceğiz. Örneğin, bir `SmsNotifier` veya bir `EmailNotifier` enjekte edebiliriz. Fatura kesicimiz artık bilgilendirme implementasyonu ile ilgilenmeyecek, sadece bilgilendirme sözleşmesiyle ilgilenecek. Bir sınıf, sözleşmesine (interface) riayet ettiği sürece, fatura kesicimiz bu sınıfı memnuniyetle kabul edecektir. Üstelik, sadece esneklik elde etmekle kalmayacağız, şimdi bir taklit `BillingNotifierInterface` enjekte etmek suretiyle fatura kesicimizi bilgilendiricilerden izole bir şekilde test edebileceğiz.

Interface (Arayüz) Olun

Arayüz yazılması fazladan birçok iş gibi görünebilir, bunlar gerçekte geliştirmenizi daha hızlı hale getirebilirler. Tek bir satır implementation yazmadan önce uygulamanızın tüm back-end'ini mock ve test etmek için interface'leri kullanın!

Peki, bağımlılık enjeksiyonunu nasıl *yaparız*? Cevabı basit:

```
$biller = new StripeBiller(new SmsNotifier);
```

İşte bağımlılık enjeksiyonu. Fatura kesicinin kullanıcıların bilgilendirilmesiyle ilgileniyor olması yerine, ona sadece bir bilgilendirici geçiyoruz. Bu basit değişiklik, uygulamanız için inanılmaz şeyler yapabilecektir. Sınıf sorumlulukları açıkça betimlendiği için, kodunuz anında daha sürdürülebilir hale gelir. Ayrıca, test altındaki kodunuzu izole etmek için taklit bağımlılıkları kolaylıkla enjekte edebileceğiniz için test edilebilirlik de yükseklere fırlayacaktır.

Fakat IoC konteynerlerinden ne haber? Bağımlılık enjeksiyonu yapmak için bunlar gerekmiyor mu? Kesinlikle hayır! Sonraki bölümlerde göreceğiniz üzere, konteynerler bağımlılık enjeksiyonu

yönetimini kolaylaştırırlar ama bir gereklilik değildirler. Bu bölümdeki ilkeleri takip etmek suretiyle, bir konteyneriniz olup olmadığına bakmaksızın herhangi bir projenizde bağımlılık enjeksiyonu uygulayabilirsiniz.

Çok Mu Java Tarzı?

PHP'de interface kullanımına sık yapılan bir eleştiri, kodunuzu "Java"ya çok benzer bir hale getirdiğidir. Bu insanların demek istediği, kodu çok gereksiz şeylerle dolduracağıdır. Bir interface ve bir implementation tanımlamak zorundasınız, bu da fazladan tuş dokunuşları demektir.

Küçük, basit uygulamalar için bu eleştiri belki geçerli olabilir. Interface'ler bu uygulamalar için çoğu keresinde gereksizdir ve kendinizi sadece değişmeyeceğini bildiğiniz bir implementasyona bağlamaya "TAMAM" diyebiliriz. İmplementasyonunuzun değişmeyeceğinden eminseniz, interface kullanmanıza gerek yoktur. Mimari astronotları "asla emin olamayacağınızı" söylüyorlar. Fakat, gelin yüzleşelim, bazen emin olabilirsiniz.

Büyük uygulamalar için, interface'ler çok yardımcıdırlar ve kazanacağınız esneklik ve test edilebilirlikle kıyaslandığında fazladan bir şeyler yazmış olma çok sönük kalır. Bir sözleşmenin implementasyonlarının çok çabuk takas edebilebilmesi yöneticinize "vay be" dedirtecek ve değişikliklere kolayca adapte olabilen kodlar yazmanıza imkan verecektir.

Böylece, sonuç olarak, bu kitabın çok "saf" bir mimari sunduğunu aklınızda tutun. Şayet onu tekrar küçük bir uygulama için ölçeklendirmeniz gerekirse, suçluluk hissetmeyin. Unutmayın, biz hepimiz "mutlu kodlar" için çalışıyoruz. Yaptığınız şeyden zevk almıyorsanız veya programlamanızdan suçluluk duyuyorsanız, durun ve yeniden değerlendirin.