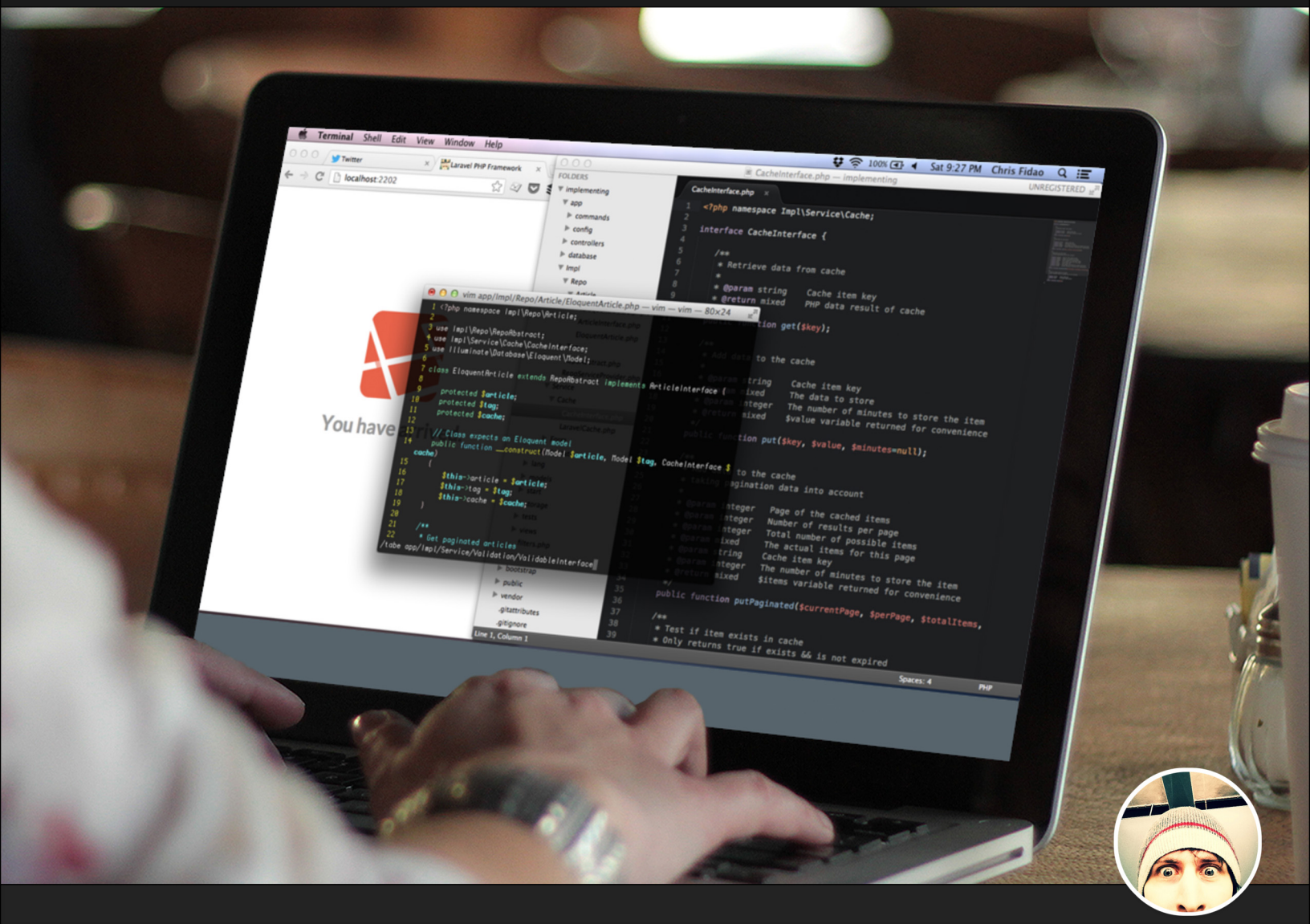

Implementing Laravel

YAZARLAR CHRIS FIDAO **&** SİNAN ELDEM

Implementing Laravel (TR) Türkçe Çevirisi

Chris Fidao ve Sinan Eldem

Bu kitap şu adreste satılmaktadır http://leanpub.com/implementinglaravel-tr

Bu versiyon şu tarihte yayımlandı 2013-09-27

Kitabı tweetleyin!

Chris Fidao ve Sinan Eldem'a kitabını şu adresten Twitter tanıtarak yardımcı olun!

Kitap için önerilen tweet:

Implementing Laravel Türkçe Çevirisi http://leanpub.com/implementinglaravel-tr #implementingLaravelTr #laravel @laraveltr

Kitap için önerilen hashtag #implementingLaravelTr.

Kitap için diğerleri ne demiş merak ediyorsanız bağlantıya tıklayarak hashtagları arayabilirsiniz:

https://twitter.com/search/#implementingLaravelTr

İçindekiler

Ana Kavramlar

Bu kitap boyunca, Laravel'in en güçlü özelliklerinin bir kısmından yararlanacağız.

Konuya geçmeden önce, en azından Laravel'in konteynerinin ve onun Bağımlılık Enjeksiyonu kullanımını bizim için ne kadar kolaylaştırdığının bilinmesi önemlidir.

Bu bölüm Laravel'in konteynerini, "Inversion of Control" kullanımını ve Bağımlılık Enjeksiyonunu kapsayacak.

Konteyner

`Illuminate\Foundation\Application` sınıfı bütün Laravel'i birbirine bağlar. Bu sınıf bir *konteynerdir* - veriler, nesneler, sınıflar ve hatta closure'lar "içerebilir".

Temel Kullanım

Konteynerin nasıl çalıştığını görmek için, routes dosyamızda bir eksersiz üzerinden çalışalım.

Laravel'in konteyneri `ArrayAccess` interface'ini implemente eder ve bu yüzden ona bir dizi gibi erişebileceğimizi biliyoruz. Ona ilişkisel bir dizi gibi nasıl ulaşabileceğimizi görelim.

Dosya: app/routes.php

```
1  Route::get('/container', function()
2  {
3          // Application olgusunu elde et
4          $app = App::getFacadeRoot();
5
6          $app['bir_dizi'] = array('foo' => 'bar');
7
8          var_dump($app['bir_dizi']);
9  });
```

`/container` rotasına gittiğimizde, şu sonucu alacağız:

```
1  array (size=1)
2          'foo' => string 'bar' (length=3)
```

Yani, `Application`un nitelikleri ve metodları olan bir sınıf olmakla birlikte, aynı zamanda bir dizi olarak ulaşılabilir olduğunu da görüyoruz!

Facade'lar

`App::getFacadeRoot()` metodunun ne yaptığı kafanızı karıştırdı mı? `App` sınıfı bir Facade'dır. Bu bize onu her yerde kullanma, ona statik bir tarzda erişme imkanı verir. Buna karşın gerçekte static bir sınıf değildir. `getFacadeRoot` metodu sınıfın gerçek olgusunu getirecektir ki bu örnekte onu bir dizi olarak kullanabilmemiz için ihtiyacımız olan şeydir.

`Illuminate\Support\Facades` aduzayındaki bu ve diğer Facade'lara bakın.

Daha İlerisi

Şimdi konteyneri biraz süsleyelim ve bir closure atayalım:

Dosya: app/routes.php

```
Route::get('/container', function()
{
        // Application olgusunu elde et
        $app = App::getFacadeRoot();

        $app['selam_ver'] = function()
        {
                return "Merhaba, Dünya!";
        };

        return $app['selam_ver'];
});
```

`/container` rotamızı tekrar çalıştıralım ve şunu göreceğiz:

```
Merhaba, Dünya!
```

Bu görünüşte basit olsa da gerçekte oldukça güçlüdür. Bu, aslında, ayrı Illuminate paketlerinin Laravel frameworkü oluşturmak için bir diğeriyle nasıl etkileştiğinin temelidir.

İlerde, çeşitli Illuminate paketleri arasında zamk gibi etkiyen Service Providerlarının öğeleri konteynere nasıl bağladığını göreceğiz.

Inversion of Control (Devrik Kontrol)

Laravel'in `Container` sınıfı sadece bir dizi kılığına bürünmekten çok daha fazlasına sahiptir. Aynı zamanda bir Inversion of Control (IoC) konteyneri olarak da işlev görebilir.

Inversion of Control, bizim bir sınıf veya interface'i nasıl implemente ettiğimizi tanımlamamız için bir tekniktir. Örneğin, uygulamamızın eğer bir `FalanInterface` bağımlılığı varsa ve biz onu implemente eden bir `SomutFalan` sınıfını kullanmak istiyorsak, bu implementasyonu tanımladığımız yer IoC konteyneridir.

Bir kez daha `/container` rotamızı kullanarak, bunun nasıl çalıştığını basit bir örnekle görelim.

İlk olarak, birkaç sınıf - bir interface ve onu implemente eden bir sınıf - oluşturacağız. Basit olması bakımından doğruca `app/routes.php` dosyasına gidebiliriz:

Dosya: app/routes.php

```php
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

    public function greet()
    {
        return 'Merhaba, Dünya!';
    }
}
```

Şimdi, bu sınıfları kulanmak için konteynerimizle ne yapabileceğimizi görelim. Önce "binding (bağlama)" kavramına gireceğiz.

Dosya: app/routes.php

```php
Route::get('/container', function()
{
    // Application olgusunu elde et
    $app = App::getFacadeRoot();

    $app->bind('GreetableInterface', function()
    {
        return new HelloWorld;
    });

    $greeter = $app->make('GreetableInterface');

    return $greeter->greet();
});
```

Dizi şeklinde erişilebilir `$app['GreetableInterface']` kullanmak yerine `bind()` metodunu kullandık.

Bu, Laravel'in IoC konteynerinin, `GreetableInterface` istenen her zamanda `HelloWorld` sınıfının döndürülmesi amacıyla kullanılmasıdır.

Bu yolla, implementasyonları "takas" edebiliyoruz! Örneğin, `HelloWorld` yerine bir `GoodbyeCruelWorld` implementasyonu yapabilirim ve `GreetableInterface` istendiği zaman konteynerin bunu döndürmesine karar verebilirim.

Bu, uygulamamızda sürdürülebilirliğe götürür. Konteyneri kullanarak, bir konumdaki implementasyonlarımızı, uygulama kodumuzun diğer alanlarını etkilemeksizin (ideal olarak) takas edebiliriz.

Gerçek Dünya Kullanımı

Tüm bu bağlamalarımızı uygulamanızda nereye koyacaksınız? Eğer `start.php`, `filters.php`, `routes.php` ve diğer bootstrap dosyalarınızı bağlamalarla doldurup karıştırmak istemiyorsanız, bu durumda Service Provider (Hizmet Sağlayıcı) sınıflarını kullanabilirsiniz.

Service Providerları özel olarak Laravel'in konteynerine bağlamaları kayda geçirmek için oluşturulurlar. Aslında, neredeyse tüm Illuminate paketleri sadece bu işi yapan bir Service Provider kullanmaktadır.

Hizmet Sağlayıcılarının bir `Illuminate` paketi içinde nasıl kullanıldığının bir örneğini görelim. Pagination paketini inceleyeceğiz.

Öncelikle, Pagination Service Provider'in `register()` metodu şöyledir:

Illuminate\Pagination\PaginationServiceProvider.php

```php
public function register()
{
    $this->app['paginator'] = $this->app->share(function($app)
    {
        $paginator = new Environment(
            $app['request'],
            $app['view'],
            $app['translator']
        );

        $paginator->setViewName(
            $app['config']['view.pagination']
        );

        return $paginator;
    });
}
```

Bu `register()` metodu `app/config/app.php` dosyası içinde belirtilen her Service Provider'ı üzerinde otomatik olarak çağrılmaktadır.

Pekiyi, bu `register()` metodunda neler yapılıyor? Birincisi ve en önemlisi, "paginator" olgusunu konteynere kayda geçiriyor. Bu, uygulamanın her yerinde `$app['paginator']` ve `App::make('paginator')` kulllanılabilmesini sağlıyor.

Daha sonra, tıpkı 'selam_ver' örneğinde yaptığımız gibi, bir closure'un sonucu olarak döndürülmek üzere 'paginator' olgusunu tanımlıyor.

`$this->app->share()` kullanılması kafanızı karıştırmasın. Share metodu sadece closure'un bir singleton olarak kullanılması için bir yol sağlar, `$this->app->instance('paginator', new Environment)` çağırmak gibi bir şeydir.

Bu closure yeni bir `Pagination\Environment` nesnesi oluşturuyor, onun üzerinde bir yapılandırma değeri ayarlıyor ve onu döndürüyor.

Bu Hizmet Sağlayıcının diğer uygulama bağlamalarını kullandığı dikkatinizi çekmiş olmalı! `Pagination\Environmer` sınıfı, oluşturucu metodunda açıkça bazı bağımlılıklar alıyor - bir request nesnesi `$app['request']`, bir view nesnesi `$app['view']` ve bir translator `$app['translator']`. Neyse ki, bu bağlamalar Illuminate'in diğer paketlerinde oluşturulmuş olan çeşitli Service Providerlarda tanımlanmışlardır.

Çeşitli Illuminate paketlerinin birbirleriyle nasıl etkileşebildiğini de görebiliyoruz. Onlar uygulama konteynerine bağlanmış oldukları için, diğer paketlerde (veya kendi kodumuz içinde!) onları kullanabiliyoruz ve bizim kodumuzu spesifik bir sınıfa gerçekten bağlamamamız gerekmiyor.

Bağımlılık Enjeksiyonu

Konteynerin nasıl çalıştığını gördüğümüze göre, Laravel'de Bağımlılık Enjeksiyonunu uygulamak için onu nasıl kullanabileceğimize bakabiliriz.

Bağımlılık Enjeksiyonu Nedir?

Bağımlılık Enjeksiyonu bir sınıf bağımlılığını sınıf kodunun kendisi içindeki bir yerlerde başlatmak yerine, sınıf içine eklenmesi (enjekte edilmesi) eylemidir. Sıklıkla, bağımlılıklar bir oluşturucu metodun type-hinted (tip dayatmalı) parametreleri olarak tanımlanırlar.

Örneğin şu oluşturucu metodu ele alalım:

```
1  public function __construct(HelloWorld $greeter)
2  {
3      $this->greeter = $greeter;
4  }
```

Bir parametre olarak `HelloWorld` tip dayatması yapmakla, bir `HelloWorld` olgusunun sınıfımızın bir bağımlılığı olduğunu açıkça ifade ediyoruz.

Bu, direkt olgu başlatmanın karşıtıdır:

```
1  public function __construct()
2  {
3      $this->greeter = new HelloWorld;
4  }
```

Kendi kendinize *neden* Bağımlılık Enjeksiyonu kullanılıyor diye soruyorsanız, bu Stack Overflow cevabı[1] başlamak için harika bir yerdir. Aşağıdaki örneklerde onun bazı yararlarını anlatacağım.

Sonra da Laravel'in IoC konteynerini kullanan Bağımlılık Enjeksiyonu örneğini iş başında göreceğiz.

[1] http://stackoverflow.com/questions/130794/what-is-dependency-injection

Controller Bağımlılıklarının Eklenmesi

Bu Laravelde çok sık kullanılan bir durumdur.

Normalde, bir controlleri oluşturucu metodunda bir sınıf bekleyecek şekilde ayarlarsak, bu sınıf oluşturulurken bağımlılıklarını da eklememiz gerekir. Ancak, bir Laravel controllerinde bir bağımlılık tanımladığımızda ne olur? Controlleri kendimiz başka bir yerde başlatmamız gerekecektir:

```
1  $crtl = new ContainerController( new HelloWorld );
```

Bu harika, ancak Laravel'de bir controlleri direkt olarak başlatmayız - bunu bizim için router halleder.

Bununla birlikte, biz yine de Laravel'in IoC konteynerini kullanmak suretiyle controller bağımlılıklarını enjekte edebiliriz!

Daha önce kullandığımız aynı `GreetableInterface` ve `HelloWorld` sınıflarını kullanarak, şimdi `/container` rotamızı bir controllere bağlamayı düşünelim:

Dosya: app/routes.php

```
1  interface GreetableInterface {
2
3      public function greet();
4
5  }
6
7  class HelloWorld implements GreetableInterface {
8
9      public function greet()
10     {
11         return 'Merhaba, Dünya!';
12     }
13 }
14
15 Route::get('/container', 'ContainerController@container);
```

Şimdi de yeni controllerimizde, oluşturucu metodunda bir parametre olarak `HelloWorld` ayarlayabiliriz:

Dosya: app/controllers/ContainerController.php

```php
<?php

class ContainerController extends BaseController {

    protected $greeter;

    // Sınıf bağımlılığı: HelloWorld
    public function __construct(HelloWorld $greeter)
    {
        $this->greeter = $greeter;
    }

    public function container()
    {
        return $this->greeter->greet();
    }

}
```

Şimdi `/container` rotanıza gidin ve yine şunu göreceksiniz:

```
Merhaba, Dünya!
```

Ancak, dikkat ediniz, konteynere hiçbir şey BAĞLAMADIK. O sadece, controllere geçmiş olduğumuz `HelloWorld`un bir olgusunu "çalıştırdı"!

Bunun nedeni IoC konteynerinin bir controllerin oluşturucu metodunda ayarlanan herhangi bir bağımlılığı otomatik olarak çözümlemeye çalışmasıdır. Laravel belirtilen bağımlılığı bizim için enjekte edecektir!

Bağımlılıklar Olarak Interface'ler

Daha bitirmedik ama. Ne yapacağımızı şimdi görün!

Bir controller'in bağımlılığı olarak `HelloWorld` sınıfını belirtmek yerine `GreetableInterface` interface'ini belirtsek ne olacaktı ?

Controller kodunu şöyle yapsak:

Dosya: app/controllers/ContainerController.php

```php
<?php

class ContainerController extends BaseController {

    protected $greeter;

    // Sınıf bağımlılığı: GreetableInterface
    public function __construct(GreetableInterface $greeter)
    {
        $this->greeter = $greeter;
    }

    public function container()
    {
        echo $this->greeter->greet();
    }

}
```

Bunu olduğu gibi çalıştırmayı denersek bir hata alırız:

```
Illuminate\Container\BindingResolutionException:
Target [GreetableInterface] is not instantiable (Hedef [GreetableInterface] başla\
tılamadı)
```

`GreetableInterface` sınıfı tabii ki başlatılamaz çünkü o bir interface'dir. Bununla birlikte, Laravel'in sınıf bağımlılığını çözmek için onu başlatma çabasına girdiğini görebiliyoruz.

Bunu düzeltelim - controller'imiz bir `GreetableInterface` olgusuna bağımlı olduğunu hissettiği zaman Laravel'in controller'e `HelloWorld` olgusu vermesi için konteynerin `bind()` metodunu kullanacağız:

Dosya: app/routes.php

```php
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

    public function greet()
    {
        return 'Merhaba, Dünya!';
    }
}

// GreetableInterface istendiğinde
// HelloWorld bağlamasını burada yapıyoruz!!
App::bind('GreetableInterface', 'HelloWorld');

Route::get('/container', 'ContainerController@container);
```

Şimdi `/container` rotanızı tekrar çalıştırırsanız, aynı şekilde yine `Merhaba, Dünya!` göreceksiniz!

Dikkat ediniz, `HelloWorld` bağlamak için bir closure kullanmadık - İstediğiniz bir somut sınıfı basitçe bir string şeklinde geçebiliyorsunuz. Implentasyonunuzun kendi oluşturucu metoduna geçilmesi gereken kendi bağımlılıkları olduğu takdirde bir closure yararlı olacaktır.

Neden Bağımlılık Enjeksiyonu?

Bir bağımlılık olarak somut bir sınıf yerine neden bir interface belirtmek istiyoruz ki?

Oluşturucuya verilen bir sınıf bağımlılığının bir interface'in bir alt sınıfı olabilmesi için bunu istiyoruz. Bu sayede, - ihtiyacımız olan metod her zaman mevcut olacak - herhangi bir implementasyonu güvenle kullanabiliyoruz.

Özlü bir ifadeyle, **uygulama kodumuzun diğer kısımlarını etkilemeksizin ilgili implementasyonu değiştirebiliriz**.

İşte bir örnek. Gerçek uygulamalarda birçok kez yapmak zorunda kaldığım bir şey.

Bu örneği kopyala yapıştır yapmayın. Konuyu temiz tutmak amacıyla, API keyleri için yapılandırma değişkenleri kullanımı gibi bazı detayları atladım çünkü.

Diyelim ki, uygulamamız Amazon'un AWS'sini kullanarak emailler gönderiyor. Bunu gerçekleştirmek için, bir `Emailer` interface ve bunu implemente eden bir `AwsEmailer` sınıfı tanımladık:

```
interface Emailer {

    public function send($to, $from, $subject, $message);
}

class AwsEmailer implements Emailer {

    protected $aws;

    public function __construct(AwsSDK $aws)
    {
        $this->aws = $aws;
    }

    public function send($to, $from, $subject, $message)
    {
        $this->aws->addTo($to)
            ->setFrom($from)
            ->setSubject($subject)
            ->setMessage($message);
            ->sendEmail();
    }
}
```

`Emailere` `AwsEmailer` implementasyonunu bağlıyoruz:

```
App::bind('Emailer', function()
{
    return new AwsEmailer( new AwsSDK );
});
```

Bir controller bağımlılık olarak `Emailer` interface'ini kullanıyor:

Dosya: app/controllers/EmailController.php

```
class EmailController extends BaseController {

    protected $emailer;

    // Sınıf bağımlılığı: Emailer
    public function __construct(Emailer $emailer)
    {
        $this->emailer = $emailer;
    }

    public function email()
    {
        $this->emailer->send(
            'ex-to@example.com',
            'ex-from@example.com',
            'Peanut Butter Jelly Time!',
            "It's that time again! And so on!"
        );

        return Redirect::to('/');
    }

}
```

Bir süre geçtikten sonra uygulamamızın kapsamı büyüdü ve AWS'nin sağladığından daha fonksiyonel bir şeye ihtiyaç doğdu. Biraz araştırdıktan ve seçenekleri değerlendirdikten sonra SendGrid üzerinde karar kıldık.

Uygulamamızı SendGrid kullanacak şekilde değiştirmek için neler yapmamız gerekiyor? Interface'ler ve Laravel'in IoC konteynerini kullandığımız için, SendGrid'e geçiş çok kolaydır!

İlk olarak, `Emailer` interface'inin SendGrid kullanan bir implementasyonunu yapacağız!

```php
class SendGridEmailer implements Emailer {

    protected $sendgrid;

    public function __construct(SendGridSDK $sendgrid)
    {
        $this->sendgrid = $sendgrid;
    }

    public function send($to, $from, $subject, $message)
    {
        $mail = $this->sendgrid->mail->instance();

        $mail->addTo($to)
            ->setFrom($from)
            ->setSubject($subject)
            ->setText( strip_tags($message) )
            ->setHtml($message)
            ->send();

        $this->sendgrid->web->send($mail);
    }
}
```

Sonra da, (ve son olarak!), uygulamamızı Aws yerine SendGrid kullanacak şekilde ayarlayacağız. IoC konteynerinde `bind()` metoduna bir çağrımız olması nedeniyle, yapacağımız *tek* değişiklik `Emailer`in implementasyonunu `AwsEmailer`den `SendGridEmailer`e değiştirmektir:

```php
// Eskisi
App::bind('Emailer', function()
{
    return new AwsEmailer( new AwsSDK );
});

// Yenisi
App::bind('Emailer', function()
{
    return new SendGridEmailer( new SendGridSDK );
});
```

Dikkat ederseniz tüm bunları, uygulamamızın başka yerlerindeki kodun tek bir satırını değiştirmeksizin yaptık. Bir bağımlılık olarak `Emailer` interface kullanımına zorlamakla, enjekte edilecek bir sınıfta `send()` metodunun mevcut olması garanti altına alınmış oluyor.

Örneğimizde bunu görebiliyoruz. Implementasyonu `AwsEmailer`den `SendGridEmailere` değiştirdiğimiz zaman, controller bir değişiklik yapılmasına gerek kalmadan hala `$this->emailer->send()` metodunu çağırmaktadır.

Özet

Bağımlılık Enjeksiyonu ve Inversion of Control, Laravel geliştirmede tekrar tekrar kullanılan desenlerdir.

Gördüğünüz gibi, kodumuzu daha sürdürülebilir yapmak ve test edilebilirliğe yardımcı olması için birçok interface tanımlayacağız. Laravel'in IoC konteyneri bizim için bunu kolay bir hale getirmektedir.