


# De Aprendiz a Artesão

## Arquitetura de Aplicação Avançada com Laravel 4

Por Taylor Otwell

# Laravel: De Aprendiz a Artesão (Brazilian Portuguese)

Taylor Otwell and Pedro Borges

This book is for sale at http://leanpub.com/laravel-pt-br

This version was published on 2013-10-01

# Tweet This Book!

Please help Taylor Otwell and Pedro Borges by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Acabei de comprar o livro ”Laravel: De Aprendiz a Artesão” por @taylorotwell.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#

# Conteúdo

**Injeção de Dependência** . . . . . **1**
O Problema . . . . . 1
Construa um Contrato . . . . . 2
Indo Além . . . . . 4
Não é muito Java? . . . . . 6

# Injeção de Dependência

## O Problema

A fundação do *framework* Laravel é seu poderoso container de *inversão de controle* (*inversion of control* ou IoC, em inglês). Para compreender o *framework* de verdade, é necessário ter um bom entendimento sobre o funcionamento do container. Entretanto, nós devemos notar que o container IoC é simplesmente um mecanismo conveniente para se alcançar o padrão de desenvolvimento de programas *injeção de dependência* (*dependency injection*, em inglês). O uso do container não é obrigatório para se injetar dependências, ele apenas facilita esta tarefa.

Em primeiro lugar, vamos explorar porque a injeção de dependência é vantajosa. Considere a seguinte classe e método:

```
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 }
```

Mesmo sendo um código conciso, não podemos testá-lo sem acessar um banco de dados. Em outras palavras, o ORM Eloquent está *fortemente acoplado* ao nosso controlador. Não podemos, de forma alguma, usar ou testar este controlador sem usar também todo o Eloquent, incluindo a necessidade de um banco de dados. Este código também viola um princípio de desenvolvimento de programas conhecido como *separação de conceitos* (*separation of concerns* ou SoC, em inglês). Em outras palavras: nosso controlador sabe mais do que deveria. Controladores não precisam saber de *onde* os dados vêm, mas somente como acessá-los. O controlador não precisa saber que os dados estão disponíveis em MySQL, apenas que eles existem em algum *lugar*.

### Separação de conceitos

Toda classe deve ter apenas uma responsabilidade e esta responsabilidade deve ser completamente encapsulada pela classe.

Sendo assim, é melhor desacoplar completamente nossa camada *web* (controlador) da nossa camada de acesso aos dados. Isso permitirá migrar mais facilmente nossa implementação de

armazenamento de dados e tornará o nosso código mais fácil de ser testado. Pense na *"web"* apenas como uma camada de transporte para a sua aplicação "real".

Imagine que sua aplicação é um monitor com portas para diversos cabos. Você pode acessar as funcionalidades do monitor via HDMI, VGA ou DVI. Pense na internet como sendo apenas um cabo conectado à sua aplicação. O monitor funciona independente do cabo utilizado. O cabo é apenas um meio de transporte, assim como HTTP é um meio de transporte que leva à sua aplicação. Assim sendo, nós não queremos entupir nosso meio de transporte (o controlador) com a parte lógica da aplicação. Isso permitirá que qualquer camada de transporte, tais como uma API ou aplicação móvel, acessem a lógica da nossa aplicação.

Por isso, ao invés de acoplar o controlador ao Eloquent, vamos injetar uma classe repositória.

## Construa um Contrato

Em primeiro lugar, vamos definir uma interface e uma implementação correspondente:

```
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 }
```

Em seguida, vamos injetar uma implementação desta interface em nosso controlador:

```
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'));
        }

}
```

Nosso controlador é completamente ignorante quanto ao local onde os dados estão sendo armazenados. Neste caso, esta ignorância é benéfica! Os dados podem estar em um banco de dados MySQL, MongoDB ou Redis. Nosso controlador não reconhece a diferença, isso não é sua responsabilidade. Fazendo essa pequena mudança, nós podemos testar nossa camada *web* separadamente da nossa camada de dados, além de podermos facilmente alternar nossa implementação de armazenamento de dados.

## Respeite os limites

Lembre-se de respeitar os limites da responsabilidade. Controladores e rotas servem como mediadores entre HTTP e sua aplicação. Ao escrever uma aplicação de grande porte, não polua-os com lógica de domínio.

Para solidificar nossa compreensão, vamos escrever uma teste rápido. Em primeiro lugar, vamos simular (*mock*, em inglês) o repositório vinculando-o ao container IoC da aplicação. Em seguida, nos certificaremos de que o controlador invoca o repositório devidamente:

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

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

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

### Faça de conta

Neste exemplo, nós usamos uma biblioteca chamada `Mockery`. Esta biblioteca oferece uma interface limpa e expressiva para fazer os *mocks* das suas classes. Mockery pode ser facilmente instalado via Composer.

## Indo Além

Vamos considerar um outro exemplo para solidificar nossa compreensão. Suponha que nós queremos notificar nossos clientes sobre as cobranças realizadas em suas contas. Para isso, vamos definir duas interfaces, ou contratos. Estes contratos nos darão flexibilidade para mudar suas implementações no futuro.

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

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

Continuando, vamos construir uma implementação do nosso contrato chamado `BillerInterface`:

```
class StripeBiller implements BillerInterface {

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

        public function bill(array $user, $amount)
        {
                // Cobrar o usuário via Stripe...

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

}
```

Porque separamos a responsabilidade de cada classe, agora nós podemos facilmente injetar várias implementações de notificação em nossa classe de cobrança. Por exemplo, nós poderíamos injetar um `SmsNotifier` ou um `EmailNotifier`. A cobrança não está mais preocupado com a implementação da notificação, somente com o contrato. Enquanto uma classe estiver de acordo com o contrato (interface), a cobrança irá aceitá-la. Além do mais, nós não apenas adicionamos flexibilidade, mas também a possibilidade de testarmos a nossa cobrança separadamente dos notificadores apenas injetando um *mock* `BillingNotifierInterface`.

## Use interfaces

Escrever interfaces pode parecer muito trabalho extra, mas na verdade, elas tornam o seu desenvolvimento mais rápido. Use interfaces para simular e testar todo o *back-end* da sua aplicação antes de escrever uma única linha de implementação!

Então, como nós *podemos* injetar uma dependência? É simples assim:

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

Isso é injeção de dependência. Ao invés da cobrança se preocupar em notificar os usuários, nós simplesmente lhe passamos um notificador. Uma mudança simples como esta pode fazer maravilhas à sua aplicação. Seu código instantaneamente se torna mais fácil de manter, porque as responsabilidades de cada classe foram claramente definidas. A testabilidade de suas aplicações aumentará consideravelmente porque agora você pode injetar *mocks* de dependências para isolar o código em teste.

Mas, e quanto aos containers IoC? Eles não são necessários na injeção de dependência? Claro que não! Conforme veremos nos próximos capítulos, containers tornam a injeção de dependência mais fácil de gerenciar, mas o seu uso não é obrigatório. Seguindo os princípios deste capítulo, você já pode praticar injeção de dependência em qualquer projeto, mesmo que você ainda não tenha um container à sua disposição.

# Não é muito Java?

Uma crítica muito comum do uso de interfaces em PHP é que elas tornam o seu código muito parecido com o "Java". Em outras palavras, o código se torna muito verbal. Você precisa definir uma interface e uma implementação; isso exigirá algumas "tecladas" a mais.

Para aplicações simples e menores, esta crítica pode até ser válida. Muitas vezes, as interfaces não são necessárias nestas aplicações e é perfeitamente "ok" não usá-las. Se você tem certeza que a implementação não mudará, você não precisa criar uma interface.

Já para aplicações maiores, as interfaces serão muito úteis. As tecladas extras não serão nada em comparação com a flexibilidade e testabilidade que você ganhará. Poder mudar rapidamente a implementação de uma contrato arrancará um "uau" do seu chefe, além de permitir que você escreva um código que facilmente se adapta a mudanças.

Para concluir, tenha em mente que este livro apresenta uma arquitetura muito "pura". Caso você precise reduzí-la para uma aplicação menor, não sinta-se culpado. Lembre-se, você está tentando "programar com alegria". Se você não gosta do que faz ou está programando por culpa, pare um pouco e faça uma reavaliação.