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

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 Taylor Otwell and Pedro Borges

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

Nota do Autor

Desde que criei o *framework* Laravel, tenho recebido inúmeros pedidos de um livro com instruções que ajudem na construção de aplicações bem estruturadas e complexas. Como cada aplicação é única, tal livro deve conter conselhos gerais, mas que ao mesmo tempo são práticos e podem ser facilmente aplicados a uma infinidade de projetos.

Assim, começaremos cobrindo os elementos fundamentais da injeção de dependência. Em seguida, analisaremos a fundo os provedores de serviços e estruturas de aplicação. Também veremos rapidamente sobre os princípios SOLID. Um profundo conhecimento destes tópicos lhe dará uma base sólida para todos os seus projetos no Laravel.

Caso você tenha alguma pergunta sobre arquitetura avançada no Laravel ou queira que eu adicione algo ao livro, por favor me escreva. Eu planejo expandir este livro baseado na opinião da comunidade. Suas ideias são importantes!

Para finalizar, muito obrigado por fazer parte da comunidade Laravel. Todos vocês ajudaram a tornar o desenvolvimento em PHP mais gostoso e prazeroso a milhares de pessoas ao redor do mundo. Programe com alegria!

Taylor Otwell

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:

```
1  class UserController extends BaseController {
2
3          public function __construct(UserRepositoryInterface $users)
4          {
5                  $this->users = $users;
6          }
7
8          public function getIndex()
9          {
10                 $users = $this->users->all();
11
12                 return View::make('users.index', compact('users'));
13         }
14
15 }
```

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.

Container IoC

Vinculação Básica

Agora que já aprendemos sobre a injeção de dependência, vamos explorar os containers de inversão de controle (IoC). Containers IoC tornam o gerenciamento de suas classes muito mais conveniente e o Laravel possui um container muito poderoso. O container IoC é a peça central do *framework* Laravel, sendo o responsável por todos os componentes do *framework* funcionarem juntos. Na realidade, a classe `Application` no Laravel extende a classe `Container`!

Container IoC

Containers de inversão de controle tornam a injeção de dependências mais conveniente. Uma classe ou interface é definida apenas uma vez no container e ele administra a resolução e injeção desses objetos sempre que necessário em sua aplicação.

Em uma aplicação Laravel, o container IoC pode ser acessado via a *façade* (fachada, em francês) `App`. O container possui uma infinidade de métodos, mas vamos iniciar com o básico. Seguiremos usando nossas `BillerInterface` e `BillingNotifierInterface` do capítulo anterior; assumiremos que nossa aplicação usa o serviço Stripe[1] para processar pagamentos. A implementação Stripe da interface pode ser vinculada (*bind*, em inglês) ao container da seguinte maneira:

```
1  App::bind('BillerInterface', function()
2  {
3          return new StripeBiller(App::make('BillingNotifierInterface'));
4  })
```

Note que dentro do resolvedor `BillerInterface` nós também resolvemos uma implementação da `BillingNotifierInterface`. Vamos definir este vínculo também:

```
1  App::bind('BillingNotifierInterface', function()
2  {
3          return new EmailBillingNotifier;
4  });
```

Como você pode ver, o container é um local onde podemos armazenar *closures* que resolvem várias classes. Quando uma classe é registrada no container, nós podemos resolvê-la facilmente em qualquer lugar da nossa aplicação. Nós também podemos resolver o vínculo de outros containers dentro de um resolvedor.

[1] https://stripe.com

Tem espinha?

O container IoC do Laravel é um substituto do container Pimple[2], criado por Fabien Potencier. Então, se você já está usando o Pimple em algum projeto, sinta-se à vontade e faça um *upgrade* para o componente Illuminate Container[3] para obter alguns recursos extras!

Após começarmos a usar o container, nós podemos alterar a implementação da interface mudando apenas uma linha. Como exemplo, considere o código a seguir:

```
class UserController extends BaseController {

        public function __construct(BillerInterface $biller)
        {
                $this->biller = $biller;
        }

}
```

Quando este controlador é instanciado pelo container IoC, o `StripeBiller`, que inclue o `EmailBillingNotifier`, será injetado nesta instância. Se mais tarde nós quisermos mudar a implementação do notificador, só precisaremos alterar o vínculo:

```
App::bind('BillingNotifierInterface', function()
{
        return new SmsBillingNotifier;
});
```

Agora, não importa onde o notificador é resolvido em nossa aplicação, nós sempre obteremos a implementação `SmsBillingNotifier`. Usando esta arquitetura, nossa aplicação poderá alternar implementações de vários serviços rapidamente.

A possibilidade de poder mudar as implementações de uma interface com apenas uma linha de código é realmente facinante. Por exemplo, imagine que nós queremos mudar o provedor de serviço SMS para o Twilio[4]. Nós podemos desenvolver uma nova implementação do notificador para o Twilio e alterar o vínculo. Se encontrarmos algum problema na transição para o Twilio, poderemos voltar para o provedor antigo fazendo uma simples alteração no vínculo IoC. Como você pode ver, os benefícios ao se usar a injeção de dependência vão além do que é imediatamente óbvio. Você consegue pensar em algum outro benefício do uso da injeção de dependência e do container IoC?

Às vezes, você vai querer resolver apenas uma instância de uma determinada classe em toda a sua aplicação. O método `singleton` nos ajudará nisto:

[2] https://github.com/fabpot/pimple
[3] https://github.com/illuminate/container
[4] http://twilio.com

```
App::singleton('BillingNotifierInterface', function()
{
        return new SmsBillingNotifier;
});
```

Agora, o container resolverá o notificador de cobrança uma vez e continuará usando a mesma instância sempre que a interface do notificador for solicita.

O método `instance` do container é similar ao `singleton`. Porém, nele você pode passar uma instância de um objeto já existente. Esta instância será usada sempre que o container precisar de uma instância dessa classe:

```
$notifier = new SmsBillingNotifier;

App::instance('BillingNotifierInterface', $notifier);
```

Agora que já estamos familiarizados com o básico da resolução de container usando *closures*, vamos conhecer os seus recursos mais poderosos: a habilidade de resolver classes via *reflexão*.

Container autônomo

Você está trabalhando em um projeto que não usa o *framework* Laravel? Você pode usar o container IoC do Laravel mesmo assim, basta instalar o pacote `illuminate/container` via Composer!

Resolução Reflexiva

Um dos recursos mais poderosos do container do Laravel é a sua habilidade de automaticamente resolver dependências via reflexão. Reflexão é a habilidade de inspecionar uma classe e seus métodos. Por exemplo, a classe do PHP `ReflectionClass` permite que você inspecione métodos disponíveis em uma determinada classe. O método do PHP `method_exists` também é uma forma de reflexão. Para brincar um pouco com a classe de reflexão do PHP, tente o código a seguir em uma de suas classes:

```
$reflection = new ReflectionClass('StripeBiller');

var_dump($reflection->getMethods());

var_dump($reflection->getConstants());
```

Tirando proveito deste poderoso recurso do PHP, o container IoC do Laravel pode fazer algumas coisas muito interessantes! Como exemplo, considere a seguinte classe:

```
class UserController extends BaseController {

        public function __construct(StripeBiller $biller)
        {
                $this->biller = $biller;
        }

}
```

Note que o controlador exige que o objeto passado seja uma instância da classe `StripeBiller`. Nós podemos extrair esta *indução de tipo* usando a reflexão. Quando o resolvedor para uma classe não é explicitamente vinculado ao container do Laravel, ele tentará resolver tal classe via reflexão. A ordem é a seguinte:

1. Existe um resolvedor para `StripeBiller`?
2. Não? Vamos refletir na classe `StripeBiller` para descobrir se ela possui alguma dependência.
3. Resolva qualquer dependência da classe `StripeBiller` (recursivo).
4. Crie uma nova instância de `StripeBiller` via `ReflectionClass->newInstanceArgs()`.

Como você pode perceber, o container faz toda a parte pesada do serviço, poupando você de ter que escrever resolvedores para cada uma das suas classes. Este é um dos recursos mais poderosos e únicos do container do Laravel. Entender bem como o container funciona é muito benéfico quando se está construindo uma aplicação grande.

Vamos modificar nosso controlador um pouquinho. Que tal se ele ficasse assim?

```
class UserController extends BaseController {

        public function __construct(BillerInterface $biller)
        {
                $this->biller = $biller;
        }

}
```

Assumindo que um resolvedor para a interface `BillerInterface` não foi vinculado explicitamente, como o container saberá qual classe injetar? Lembre-se, interfaces não podem ser instanciadas, elas são apenas contratos. Sem darmos mais informação ao container, ele não poderá instanciar esta dependência. Nós precisamos especificar uma classe que será usada como implementação padrão desta interface; fazemos isto através do método `bind`:

```
App::bind('BillerInterface', 'StripeBiller');
```

Aqui, nós passamos uma *string* ao invés de uma *closure*. Esta *string* dirá ao container para usar a classe `StripeBiller` sempre que ele precisar de uma implementação da interface `BillerInterface`. Novamente, nós ganhamos a habilidade de alternar as implementações de serviços mudando uma única linha no container de vinculação. Por exemplo, se quisermos usar o serviço Balanced Payments[5] para processar pagamentos, precisaremos apenas escrever uma nova implementação `BalancedBiller` da interface `BillerInterface` e mudar o nosso container desta forma:

```
App::bind('BillerInterface', 'BalancedBiller');
```

Automaticamente, esta nova implementação será usada por toda a nossa aplicação!

Você também pode usar o método `singleton` para vincular implementações às interfaces. Assim, o container irá criar apenas uma instância da classe por ciclo de *request*:

```
App::singleton('BillerInterface', 'StripeBiller');
```

Mestre em container

Deseja aprender mais sobre o container do Laravel? Leia seu código-fonte! O container é apenas uma classe: `Illuminate\Container\Container`. Leia todo o seu código para entender melhor como ele funciona debaixo do capô.

[5] https://balancedpayments.com

Interface Como Contrato

Tipagem Forte & Aves Aquáticas

Nos capítulos anteriores, nós vimos o básico da injeção de dependência: o que ela é; como ela é aplicada; e, muito dos seus benefícios. Os exemplos dados nesses capítulos também demonstraram a injeção de *interfaces* em nossas classes. Por isso, antes de avançarmos, acho bom falar mais profundamente sobre as interfaces, visto que muitos programadores de PHP não estão familiarizados como elas.

Antes de me tornar um programador de PHP, eu programava .NET. Eu gosto de sofrer ou o quê? Em .NET, interfaces estão por toda parte. Na realidade, muitas interfaces são definidas como parte do próprio núcleo do *framework* .NET, e por uma boa razão: muitas das suas linguagens, como C# e VB.NET, possuem **tipagem forte**. Em outras palavras, você precisa definir o *tipo* do objeto ou um tipo primitivo será passado para o método. Por exemplo, considere este método em C#:

```
public int BillUser(User user)
{
        this.biller.bill(user.GetId(), this.amount)
}
```

Observe que fomos obrigados a definir não somente o *tipo* dos argumentos que passaremos para o método, mas também o que o próprio método deverá retornar. C# encoraja **tipagem segura**. Não poderemos passar nada além de um objeto `User` para o método `BillUser`.

Entretanto, o PHP de modo geral usa **duck typing** (tipagem do pato). No *duck typing*, os métodos disponíveis em um objeto determinam a forma como ele pode ser usado, ao invés da sua herança ou da implementação de uma interface. Vejamos um exemplo:

```
public function billUser($user)
{
        $this->biller->bill($user->getId(), $this->amount);
}
```

Em PHP, não precisamos dizer ao método qual tipo de argumento ele deve aceitar. Podemos passar qualquer tipo, desde que tal objeto responda ao método `getId`. Este é um exemplo de *duck typing*. Se o objeto se parece com um pato, anda como um pato, grasna como um pato, ele deve ser um pato. Ou, no nosso caso, se o objeto se parece com um usuário (*user*, em inglês) e age como um usuário, ele deve ser um.

Mas, e o PHP? Ele não possui *nenhum* recurso da tipagem forte? Claro que possui! O PHP possui uma mistura de tipagem forte e *duck typing*. Para ilustrar isso, vamos reescrever nosso método `billUser`:

```
1  public function billUser(User $user)
2  {
3          $this->biller->bill($user->getId(), $amount);
4  }
```

Após induzirmos o tipo `User` em nosso método, podemos garantir que todo objeto passado para `billUser` será uma instância de `User` ou será uma instância de outra classe que extenda a classe `User`.

Há vantagens e desvantagens em ambos os sistemas de tipagem. Nas linguagens de tipagem forte, o compilador pode lhe dar um minucioso relatório de erros durante a compilação, o que é extremamente útil. As entradas e saídas dos métodos são muito mais explícitas em uma linguagem de tipagem forte.

Mas ao mesmo tempo, o caráter explícito da tipagem forte torna-a muito rígida. Por exemplo, métodos dinâmicos como o `whereEmailOrName`, oferecidos pelo Eloquent, seriam impossíveis de serem implementados em liguagens de tipagem forte como C#. Não entre em discussões acaloradas sobre qual paradigma é melhor e lembre-se, cada um possui suas vantagens e desvantagens. Não é "errado" usar a tipagem forte disponível em PHP nem é errado usar *duck typing*. Errado seria usar *exclusivamente* um, ou o outro, sem considerar o problema que precisa ser resolvido.

Um Exemplo de Contrato

Interfaces são contratos. Elas não possuem nenhum código, apenas definem alguns métodos que um objeto **deve** implementar. Se um objeto implementa uma interface, podemos ter certeza que cada método definido pela interface é válido e poderá ser invocado naquele objeto. Uma vez que o contrato garante a implementação de certos métodos, a tipagem segura torna-se mais flexível através do **polimorfismo**.

Polioquê?

Polimorfismo é uma palavra grande que essencialmente significa: uma substância que possui múltiplas formas. No contexto deste livro, é uma interface que pode ter muitas implementações. Por exemplo, `UserRepositoryInterface` pode ter uma implementação para MySQL e uma para Redis e ambas serem instâncias válidas da interface `UserRepositoryInterface`.

Para ilustrar a flexibilidade que as interfaces trazem para as linguagens de tipagem forte, vamos escrever um código simples para realizar reservas em hoteis. Considere esta interface:

```
interface ProviderInterface {
        public function getLowestPrice($location);
        public function book($location);
}
```

Quando nosso usuário reserva um quarto, nós queremos que esta ação seja registrada pelo sistema. Por isso, vamos adicionar alguns métodos à nossa classe `User`:

```
class User {

        public function bookLocation(ProviderInterface $provider, $location)
        {
                $amountCharged = $provider->book($location);

                $this->logBookedLocation($location, $amountCharged);
        }

}
```

Como estamos induzindo o tipo `ProviderInterface`, nossa classe `User` pode seguramente assumir que o método `book` sempre estará disponível. Isto nos dá a flexibilidade de podermos reutilizar o método `bookLocation` independente do hotel escolhido pelo usuário. Para finalizar, vamos endurecer esta flexibilidade:

```
$location = 'Hilton, Dallas';

$cheapestProvider = $this->findCheapest($location, array(
        new PricelineProvider,
        new OrbitzProvider,
));

$user->bookLocation($cheapestProvider, $location);
```

Maravilhoso! Não importa qual provedor possui o menor preço, nós podemos passá-lo para a instância do nosso `User` para que a reserva seja efetuada. Como a classe `User` está pedindo somente instâncias de um objeto que esteja de acordo com o contrato `ProviderInterface`, nosso código continuará funcionando mesmo quando adicionarmos novas implementações de provedores.

Esqueça os detalhes

Lembre-se, as interfaces não *fazem* nada. Elas apenas definem um conjunto de métodos que **devem** existir na classe que as implementa.

Interfaces e o Desenvolvimento em Equipe

Quando sua equipe está desenvolvendo uma aplicação grande, cada parte vai progredir em uma velocidade diferente. Por exemplo, imagine que um desenvolvedor está trabalhando na camada de dados, enquanto outro trabalha no *front-end* e na camada web (controlador). O desenvolvedor do *front-end* deseja testar seus controladores, mas o *back-end* está atrasado. Que tal se estes dois desenvolvedores concordassem em uma interface, ou contrato, que fosse implementada pelas classes do *back-end*? Algo assim:

```
interface OrderRepositoryInterface {
        public function getMostRecent(User $user);
}
```

Tendo concordado nesse contrato, o desenvolvedor do *front-end* poderá testar seus controladores, mesmo que nenhuma implementação real tenha sido escrita! Isso permitirá que os componentes de uma aplicação sejam escritos em velocidades diferentes, além de permitir que os testes também sejam devidamente escritos. E mais, esta abordagem permitirá que implementações inteiras mudem sem quebrar outras. Lembre-se, esta ignorância é benéfica. Não queremos que nossas classes saibam *como* uma dependência realiza algo, mas apenas que ela é *capaz* de realizá-lo. Assim, agora que temos um contrato definido, vamos para o controlador:

```
class OrderController {

        public function __construct(OrderRepositoryInterface $orders)
        {
                $this->orders = $orders;
        }

        public function getRecent()
        {
                $recent = $this->orders->getMostRecent(Auth::user()):

                return View::make('orders.recent', compact('recent'));
        }

}
```

O desenvolvedor do *front-end* poderia até mesmo escrever uma implementação "fictícia" da interface. Desta forma, os `views` da aplicação receberão dados fictícios:

```
class DummyOrderRepository implements OrderRepositoryInterface {

        public function getMostRecent(User $user)
        {
                return array('Order 1', 'Order 2', 'Order 3');
        }

}
```

Após ter escrito uma implementação fictícia, ela poderá ser vinculada ao container IoC e usada por toda a sua aplicação:

```
App::bind('OrderRepositoryInterface', 'DummyOrderRepository');
```

Então, quando uma implementação "real", como a `RedisOrderRepository`, for escrita pelo desenvolvedor do *back-end*, o vínculo IoC poderá ser alternado para a nova implementação e sua aplicação já estará usando os pedidos armazenados em Redis.

Interface como esquemática

Interfaces são úteis no desenvolvimento do "esqueleto" das funcionalidades providas por sua aplicação. Use-as durante o planejamento de um componente para facilitar a discussão entre sua equipe. Por exemplo, defina uma interface `BillingNotifierInterface` e converse com sua equipe sobre os métodos dela. Use a interface para concordar em uma boa API antes de começar a escrever o código de implementação!

Provedores de Serviços

Como Inicializador

Um provedor de serviço no Laravel é uma classe que registra vínculos do container IoC. O Laravel já vem com dezenas de provedores de serviços que gerenciam os vínculos dos containers dos componentes do seu núcleo. Quase todos os componentes do *framework* são registrados com um container em um provedor. A lista destes provedores usados por sua aplicação pode ser encontrada no *array* `providers` no arquivo de configuração `app/config/app.php`.

Um provedor de serviço deve possuir pelo menos um método: `register`. O método `register` é onde o provedor vincula classes ao container. Quando sua aplicação recebe uma *request* e o *framework* é inicializado, o método `register` é invocado em todos os provedores listados no arquivo de configuração. Isto acontece no começo do ciclo de vida da sua aplicação para que todos os serviços providos pelo Laravel estejam à sua disposição quando os seus próprios arquivos forem carregados, tais como aqueles na pasta `start`.

Register x Boot

Nunca tente usar um serviço no método `register`. A única tarefa deste método deve ser registrar o vínculo de objetos ao container IoC. Qualquer resolução e interação com as classes vinculadas devem ser feitas dentro do método `boot`.

Alguns pacotes de terceiros instalados via Composer já vem com um provedor de serviço. Tipicamente, as instruções de instalação destes pacotes solicitarão que você registre o provedor em sua aplicação no *array* `providers`, no arquivo de configuração. Assim que o provedor de um pacote é registrado, ele está pronto para ser usado.

Provedores de pacotes

Nem todos os pacotes de terceiros precisarão de um provedor de serviço. Na realidade, nenhum pacote *requer* um provedor, porque geralmente os provedores de serviço apenas inicializam componentes para que estes possam ser usados imediatamente. Eles são apenas um local conveniente para se organizar a inicialização e os vínculos dos containers.

Provedores Diferidos

Nem todos os provedores listados no *array* de configuração `providers` serão inicializados em todas as *requests*. Isto afetaria a performance da sua aplicação, especialmente quando esses provedores de serviços não são usados em todas as *requests*. Por exemplo, o serviço `QueueServiceProvider` não é necessário sempre, apenas quando o componente `Queue` é utilizado.

Para poder instanciar apenas os provedores que serão utilizados em uma determinada *request*, o Laravel gera uma "manifesto de serviço" e o armazena na pasta `app/storage/meta`. Este manifesto lista todos os provedores de serviços da sua aplicação, bem como os nomes dos containers de vinculação registrados por eles. Desta forma, quando sua aplicação precisa usar o componente `Queue`, o Laravel sabe que é hora de instanciar e rodar o `QueueServiceProvider`, porque ele está listado como o provedor do serviço `queue` no manifesto. Isto permite que o *framework* carregue somente os provedores de serviços necessários em cada *request*, aumentando consideravelmente a performance.

Geração do manifesto

Quando você acrescenta um novo provedor de serviço ao *array* `providers`, o Laravel gerará automaticamente um novo manifesto de serviços na próxima vez em que a sua aplicação for usada.

Quando você tiver um tempinho, dê uma olhada no manifesto de serviços para conhecer o seu conteúdo. Entender esta estrutura será útil quando você precisar depurar um provedor de serviço diferido.

Como Organizador

Uma das chaves para se construir uma aplicação Laravel bem arquitetada é aprender a usar os provedores de serviços como uma ferramenta organizacional. Quando você registra muitas classes com o container IoC nos seus arquivos `app/start`, todos estes vínculos começarão a poluí-los. Ao invés de registrar containers nesses arquivos, crie provedores de serviços para registrar serviços relacionados.

Inicializando

Os arquivos *"start"* de sua aplicação vivem na pasta `app/start`. Eles são arquivos "inicializadores" que são carregados com base no tipo de *request*. O arquivo global `start.php` é o primeiro a ser carregado, seguido pelo arquivo que possui o mesmo nome que o *environment* (ambiente). Já o arquivo *"start"* `artisan.php` é carregado sempre que um comando é executado no *console*.

Vamos explorar um exemplo. Suponha que sua aplicação esteja usando o serviço Pusher[6] para enviar (*push*, em inglês) mensagens para os clientes via *WebSockets*. Para não ficarmos presos ao Pusher, acho melhor criarmos uma interface `EventPusherInterface` e uma implementação `PusherEventPusher`. Isto nos permitirá mudar de provedor de *WebSocket* no futuro conforme nossas necessidades mudem e nossa aplicação cresça.

[6] http://pusher.com

```php
interface EventPusherInterface {
        public function push($message, array $data = array());
}

class PusherEventPusher implements EventPusherInterface {

        public function __construct(PusherSdk $pusher)
        {
                $this->pusher = $pusher;
        }

        public function push($message, array $data = array())
        {
                // Enviar a mensagem via a SDK do Pusher...
        }

}
```

Em seguida, vamos criar um provedor `EventPusherServiceProvider`:

```php
use Illuminate\Support\ServiceProvider;

class EventPusherServiceProvider extends ServiceProvider {

        public function register()
        {
                $this->app->singleton('PusherSdk', function()
                {
                        return new PusherSdk('app-key', 'secret-key');
                });

                $this->app->singleton('EventPusherInterface', 'PusherEventPusher');
        }

}
```

Excelente! Agora nós temos uma abstração para o envio de mensagens, bem como um lugar conveniente para registrar vínculos no container. Finalmente, nós só precisamos acrescentar o `EventPusherServiceProvider` ao nosso *array* `providers` no arquivo `app/config/app.php`. Agora já podemos injetar a interface `EventPusherInterface` em qualquer controlador ou classe em nossa aplicação.

Quando devo usar singleton?

Você deve avaliar se é melhor usar `bind` ou `singleton` para vincular suas classes. Caso você precise de apenas uma instância da classe por *request*, use `singleton`. Do contrário, use `bind`.

Note que o provedor de serviço possui uma instância `$app` herdada da classe base, `ServiceProvider`. Ela é uma instância completa de `Illuminate\Foundation\Application`, que extende a classe `Container`. Por isso, podemos usar todos os métodos do container IoC que já estamos acostumados. Se você preferir usar a *façade* `App` no provedor de serviço, fique à vontade:

```
1  App::singleton('EventPusherInterface', 'PusherEventPusher');
```

É claro que os provedores de serviços não estão limitados a registrarem apenas certos tipos de serviços. Nós poderíamos usá-los para registrar um serviço para armazenar arquivos na nuvem, ou outra *view engine* como o Twig, etc. Eles são apenas ferramentas inicializadoras e organizacionais para sua aplicação. Nada mais.

Então, não fique como medo de criar os seus próprios provedores de serviços. Eles não são limitados a pacotes que serão distribuídos, muito pelo contrário, são uma excelente ferramenta para ajudá-lo na organização das suas aplicações. Seja criativo e use-os para inicializar os vários componentes da sua aplicação.

Provedores de Inicialização

Os provedores de serviços serão "inicializados" durante o seu registro, em seguida, o método `boot` será executado em cada provedor. Um erro muito comum é tentar usar o serviço de outro provedor direto no método `register`. Mas, como dentro do método `register` nós não podemos ter certeza de que os demais provedores foram carregados, o serviço que você está tentando usar pode não estar disponível ainda. Por isso, sempre use o método `boot` no provedor de serviço para escrever um código que usa outros serviços. O método `register` deve ser usado **somente** para registrar serviços com o container.

Dentro do método `boot`, faça o que você quiser: registre *listeners* de eventos, inclua um arquivo com rotas, registre filtros ou o que você conseguir imaginar. Novamente, use os provedores como uma ferramenta organizacional. Você gostaria de agrupar alguns *listeners* de eventos? Colocá-los no método `boot` de um provedor de serviço é uma ótima ideia! Você também poderia incluir um arquivo PHP com "eventos" ou "rotas":

```
1  public function boot()
2  {
3          require_once __DIR__.'/events.php';
4
5          require_once __DIR__.'/routes.php';
6  }
```

Agora que já aprendemos sobre a injeção de dependência e como organizar nossos projetos usando provedores, temos uma ótima base para construir aplicações Laravel bem arquitetadas, fáceis de manter e testar. Em seguida, vamos explorar como o próprio Laravel usa os provedores e como o "motor" do *framework* funciona!

Lembre-se

Não assuma que os provedores de serviços devem ser usados somente dentro de pacotes. Crie o seu próprio provedor para ajudá-lo a organizar os serviços da sua aplicação.

Provedores do Núcleo

A essa altura, você já deve saber que sua aplicação possui muitos provedores de serviços registrados no arquivo de configuração `app.php`. Cada um desses provedores inicializa uma parte do núcleo do *framework*. Por exemplo, o provedor `MigrationServiceProvider` inicializa várias classes que executam migrações e também o comando `migrate` do Artisan. O `EventServiceProvider` inicializa e registra a classe `Dispatcher`. Alguns desses provedores são maiores do que os outros, mas cada um deles inicializa uma parte do núcleo.

Conheça seus provedores

Uma das melhores formas de aprimorar os seus conhecimentos sobre o núcleo do Laravel é lendo o código dos seus provedores de serviços. Se você já está familiarizado com as funções dos provedores de serviços e sabe o que cada provedor do núcleo registra, você terá uma compreensão muito melhor de como o *framework* Laravel funciona debaixo do capô.

A maioria dos provedores são diferidos, ou seja, eles não são carregados em todas as *requests*. Entretanto, alguns deles, como o `FilesystemServiceProvider` e o `ExceptionServiceProvider`, são carregados sempre, pois eles inicializam partes essenciais para o funcionamento do *framework*. Alguém pode até dizer que os provedores de serviços do núcleo e o container da aplicação *são* o Laravel. Eles são responsáveis por unir as muitas partes do *framework* de uma forma coesa; estes provedores formam o alicerce do *framework*.

Conforme já mencionado, se você quiser entender profundamente como o *framework* funciona, leia o código-fonte dos provedores de serviço do seu núcleo. Ao lê-lo, você compreenderá como o *framework* é formado e o que cada provedor pode oferecer à sua aplicação. Além do mais, você estará melhor preparado para contribuir para o próprio Laravel!

Estrutura de Aplicação

Introdução

"Onde devo colocar esta classe?" Esta é uma pergunta muito comum quando se usa um *framework* na criação de uma aplicação. Muitos desenvolvedores se fazem esta pergunta porque eles já ouviram que "Modelo" (*model*, em inglês) significa "Banco de Dados". Por isso, eles têm controladores que interagem com HTTP, modelos que realizam *algo* com o banco de dados e *views* que abrigam o código HTML. Mas, e as classes que enviam e-mails, validam dados e interagem com uma API para obter informações? Neste capítulo, vamos aprender como estruturar uma boa aplicação usando o *framework* Laravel, além de ver quais são alguns dos obstáculos que mais atrapalham os desenvolvedores no desenvolvimento de uma boa aplicação.

MVC está te Matando

O maior obstáculo para que os desenvolvedores alcancem um bom *design* em suas aplicações é uma sigla bem pequena: M-V-C. Modelos, *views* e controladores têm dominado o pensamento dos *frameworks* para a *web* por vários anos, em parte, devido à popularidade do *Ruby on Rails*. E, mesmo assim, peça a um desenvolvedor para definir "modelos". Normalmente, você ouvirá alguma coisa seguido pelas palavras "banco de dados". Supostamente, o modelo *é* o banco de dados. É onde *tudo* o que tem a ver com o banco de dados deve ir. Porém, você rapidamente aprenderá que sua aplicação precisa de muito mais lógica do que uma simples classe para acessar o banco de dados. Ela precisará validar dados, acessar serviços externos, enviar e-mails e muito mais.

O que é um modelo?

A palavra "modelo" tornou-se tão ambígua que acabou perdendo o seu significado. Desenvolver com um vocabulário específico nos ajudará a dividir nossa aplicação em partes menores, classes mais limpas e com uma responsabilidade bem definida.

Então, qual é a solução para este dilema? Muitos tem enchido seus controladores com lógica. Quando estes controladores se tornam enormes, eles precisam reutilizar a lógica presente em outros controladores. Ao invés de extrair essa lógica para uma outra classe, muitos desenvolvedores erroneamente assumem que eles *precisam* invocar controladores a partir de outros controladores; um padrão conhecido como "HMVC". Infelizmente, esta solução geralmente indica que uma aplicação foi mal planejada e que os controladores se tornaram muito complicados.

HMVC (geralmente) indica falta de planejamento

Já precisou invocar um controlador a partir de um outro controlador? Isso é um indicador de que a aplicação foi mal planejada e de que há muita lógica em seus controladores. Extraia essa lógica para uma terceira classe para que ela possa ser injetada em qualquer controlador.

Mas existe uma forma melhor de estruturar aplicações. Precisamos limpar das nossas mentes tudo o que já aprendemos sobre *modelos*. Na verdade, vamos excluir a pasta `models` e começar do zero!

Adeus Models

Já exluiu a pasta `models`? Ainda não? Livre-se dela logo! Vamos criar uma nova pasta dentro da pasta `app` e dar a ela o nome da nossa aplicação. Para esta discussão, vamos chamar nossa aplicação de `QuickBill` (Cobrança Rápida) e continuaremos usando algumas das interfaces e classes dos capítulos anteriores.

Lembre-se do contexto

Se você está desenvolvendo uma aplicação pequena usando o Laravel, não há nenhum problema em criar alguns modelos com o Eloquent na pasta `models`. Porém neste capítulo, estamos preocupados em descobrir uma arquitetura em "camadas" mais adequada para projetos grandes e complexos.

Assim, já devemos ter uma pasta chamada `QuickBill` na pasta da nossa aplicação, no mesmo nível das pastas `controllers` e `views`. Podemos criar quantas pastas quisermos dentro de `app/QuickBill`. Vamos criar apenas duas por enquanto: `Repositories` e `Billing`. Tendo criado estas pastas, lembre-se de registrá-las no arquivo `composer.json` para que o carregamento automático PSR-0 funcione:

```
"autoload": {
        "psr-0": {
                "QuickBill": "app/"
        }
}
```

Por enquanto, vamos deixar nossas classes Eloquent na raiz da pasta `QuickBill`. Assim, poderemos acessá-las facilmente como `QuickBill\User`, `QuickBill\Payment`, etc. A pasta `Repositories` será o lugar de classes como: `PaymentRepository` e `UserRepository`; estas devem conter todas as funções que acessam dados, tais como `getRecentPayments` e `getRichestUser`. A pasta `Billing` deverá conter todas as classes e interfaces que trabalham com outros serviços de pagamento, como Stripe e Balanced. Ao final, a estrutura desta pasta será algo assim:

```
// app
        // QuickBill
                // Repositories
                        -> UserRepository.php
                        -> PaymentRepository.php
                // Billing
                        -> BillerInterface.php
                        -> StripeBiller.php
                // Notifications
                        -> BillingNotifierInterface.php
                        -> SmsBillingNotifier.php
                User.php
                Payment.php
```

E a validação?

Onde realizar a validação é uma dúvida muito comum. Considere colocar os métodos de validação nas classes de "entidades", como: `User.php` e `Payment.php`. Alguns possíveis nomes para estes métodos seriam: `validForCreation` ou `hasValidDomain`. Alternativamente, você poderá criar uma classe `UserValidator` dentro do *namespace* `Validation` e injetá-la no seu repositório. Experimente com estas possibilidades para descobrir qual lhe agrada mais!

Eliminar a pasta `models` geralmente lhe ajudará a retirar as barreiras mentais que atrapalham um bom desenvolvimento, permitindo que você crie uma pasta com uma estrutura mais adequada à sua aplicação. Com certeza suas aplicações terão algumas semelhanças, porque toda aplicação complexa necessita de uma camada de acesso a dados (repositório), camadas de serviços externos e por aí vai.

Não tenha medo das pastas

Não tenha medo de criar outras pastas para organizar sua aplicação. Sempre divida sua aplicação em componentes menores, cada um com uma responsabilidade bem definida. Pensar além do "modelo" lhe ajudará muito. Por exemplo, você pode criar uma pasta `Repositories` para armazenar todas as classes que acessam dados.

É Tudo Sobre as Camadas

Como você já deve ter notado, a chave para uma aplicação sólida é a separação das responsabilidades, ou criação de camadas de responsabilidades. A responsabilidade dos controladores é receber *requests* HTTP e invocar as classes de "camada de negócio" apropriadas. Sua camada de negócio, ou de domínio, *é* a sua aplicação. Elas contêm as classes que acessam os dados, realizam a validação, processam os pagamentos, enviam os e-mails e qualquer outra função de sua aplicação. Na realidade, sua camada de domínio não precisa nem saber que "a *web*" existe! A

web é apenas um meio de transporte que dá acesso à sua aplicação e o seu conhecimento não deve ir além das camadas de controle (*controllers*) e de roteamento. Alcançar uma boa arquitetura é um desafio enorme, mas traz como benefícios uma grande sustentabilidade e um código limpo.

Por exemplo, ao invés de acessar uma instância da *request* da *web* na classe, nós podemos enviar a *input* do controlador para a classe. Esta simples mudança desacoplará a sua classe da *"web"* e ela poderá ser testada sem se preocupar com a simulação de uma *request*:

```
1  class BillingController extends BaseController {
2
3          public function __construct(BillerInterface $biller)
4          {
5                  $this->biller = $biller;
6          }
7
8          public function postCharge()
9          {
10                 $this->biller->chargeAccount(Auth::user(), Input::get('amount'));
11
12                 return View::make('charge.success');
13         }
14
15 }
```

Agora ficou muito mais fácil testar o método `chargeAccount`, pois ele não precisa usar as classes `Request` ou `Input` dentro da nossa implementação da `BillerInterface`. Precisamos apenas lhe dar o valor a ser cobrado como um número inteiro.

A separação das responsabilidades é uma das chaves para escrevermos aplicações sustentáveis. Pergunte-se sempre se uma classe sabe mais do que ela deveria. Pergunte-se: "Esta classe deveria de importar com *X*?" Se a resposta for "não", extraia esta lógica para outra classe e injete-a como uma dependência.

A única razão para mudar

Uma ótima forma de saber se uma classe possui reponsabilidades além do que deveria, é examinar a razão das mudanças realizadas nela. Por exemplo, nos deveríamos alterar o código de uma implementação da interface `Biller` enquanto ajustamos a lógica da notificação? Claro que não. Sua responsabilidade é realizar a cobrança e trabalhar com a lógica da notificação somente via um contrato. Manter isso em mente enquanto você programa irá ajudá-lo a identificar as áreas onde sua aplicações pode melhorar.

Onde Colocar as "Coisas"

Ao desenvolver aplicações usando o Laravel, às vezes você terá dúvidas sobre onde colocar algo. Por exemplo, onde eu devo colocar as funções auxiliares (*helpers*, em inglês)? Onde eu devo pôr

os *listeners* de eventos? E os *view composers*? Você ficará surpreso, mas a resposta é: "Onde você quiser!" O Laravel não tem muitas convenções sobre onde os arquivos devem existir. Mas, como muitas vezes esta resposta não é satisfatória, vamos explorar alguns possíveis locais para tais "coisas" antes de avançarmos.

Funções Auxiliares

O Laravel já vem com um arquivo repleto de funções auxiliares: `support/helpers.php`. Caso você queira criar um arquivo parecido com funções relevantes ao seu projeto e estilo de programação, um ótimo lugar para incluí-las é nos arquivos "start". No arquivo `start/global.php`, que é incluído em todas as *requests*, você pode requerer o seu próprio arquivo `helpers.php`:

```
// Dentro de app/start/global.php

require_once __DIR__.'/../helpers.php';
```

Listeners de Eventos

Como os *listeners* de eventos não pertencem ao arquivo `routes.php` e podem começar a poluir os arquivos "start", nós precisamos de um outro local para este código. Uma boa opção seria criar um provedor de serviço. Conforme já aprendemos, os provedores de serviços não servem apenas para registrar vinculações nos containers IoC. Eles podem ser usados para qualquer tipo de trabalho. Agrupar o registro de eventos em um provedor de serviço manterá este código perfeitamente escondindo nos bastidores da sua aplicação. *View composers*, que são um tipo de evento, também podem ser agrupados em um provedor de serviço.

Por exemplo, um provedor de serviço que registra eventos ficaria assim:

```
<?php namespace QuickBill\Providers;

use Illuminate\Support\ServiceProvider;

class BillingEventsProvider extends ServiceProvider {

        public function boot()
        {
                Event::listen('billing.failed', function($bill)
                {
                        // Fazer algo quando a cobrança falhar...
                });
        }

}
```

Após criarmos este provedor, só precisamos acrescentá-lo ao nosso *array* `providers` no arquivo de configuração `app/config/app.php`.

Por que boot?

No exemplo acima, escolhemos usar o método `boot` por um bom motivo, lembra? O método `register` em um provedor de serviço **só** deve ser usado para vincular classes ao container.

Manipulando Erros

Se sua aplicação lida com muitos erros, este código logo tomará conta do arquivos "start". Por isso, como fizemos com os eventos, é melhor mover este código para um provedor de serviço. Seu nome pode ser algo assim: `QuickBillErrorProvider`. Todo o código para manipular os erros da sua aplicação pode ser registrado no método `boot` desse provedor. Novamente, isso manterá esse tipo de código, tão comum, longe dos arquivos da sua aplicação. Vamos ver como ficaria um provedor que manipula erros:

```
1  <?php namespace QuickBill\Providers;
2
3  use App, Illuminate\Support\ServiceProvider;
4
5  class QuickBillErrorProvider extends ServiceProvider {
6
7          public function register()
8          {
9                  //
10         }
11
12         public function boot()
13         {
14                 App::error(function(BillingFailedException $e)
15                 {
16                         // Fazer algo com a exceção da falha na cobrança...
17                 });
18         }
19
20 }
```

Um pequena solução

Se você possui apenas um ou dois manipuladores de erros, é claro que a melhor solução é deixá-los nos arquivos "start".

O Resto

De forma geral, outras classes podem ser facilmente organizadas na pasta da sua aplicação usando a estrutura PSR-0. *Listeners* de eventos, manipuladores de erro e outras operações do tipo "registro" podem ser colocados em um provedor de serviço. Com o que já aprendemos até aqui, você já deve ser capaz de tomar decisões sábias sobre onde colocar qualquer tipo de código. Mas, não hesite em experimentar. A beleza do *framework* Laravel é que você pode decidir o que funciona melhor para você. Descubra a estrutura que melhor se aplica às suas aplicações. E, também, não deixe de compartilhar o que você aprender com outros desenvolvedores!

Por exemplo, como você já deve ter notado acima, você pode criar um *namespace* chamado `Providers` para todos os provedores da sua aplicação. A estrutura da pasta ficará assim:

```
// app
        // QuickBill
                // Billing
                // Extensions
                        // Pagination
                                -> Environment.php
                // Providers
                        -> EventPusherServiceProvider.php
                // Repositories
                User.php
                Payment.php
```

Observe que nós temos um *namespace* `Providers` e um outro `Extensions`. Todos os provedores de serviço da sua aplicação poderão ser armazenados na pasta `Providers`. A pasta `Extensions` é um lugar conveniente para armazenar extensões de classes do *framework* feitas por você.

Arquitetura Aplicada: Desacoplando os Manipuladores

Introdução

Agora que já discutimos os vários aspectos da arquitetura de aplicação estável usando o Laravel 4, vamos ver algo mais específico. Neste capítulo, discutiremos dicas para desacoplar diversos manipuladores, como o de *queue* e o de eventos, além de outras estruturas, como o filtro de rotas.

Não polua a camada de transporte

A maioria dos "manipuladores" pode ser considerada um componente da *camada de transporte*. Em outras palavras, eles são invocados através de *queue workers*, eventos ou *web requests*. Trate estes manipuladores como se fossem controladores, evitando entupí-los com detalhes da implementação da sua aplicação.

Desacoplando os Manipuladores

Para começar, vamos direto ao primeiro exemplo. Considere um manipulador de *queue* que envia uma mensagem SMS para um determinado usuário. Após o envio da mensagem, o manipulador registrará a mensagem para que nós possamos manter um histórico de todas as mensagens SMS enviadas para aquele usuário. Nosso código ficará assim:

```
class SendSMS {

        public function fire($job, $data)
        {
                $twilio = new Twilio_SMS($apiKey);

                $twilio->sendTextMessage(array(
                        'to' => $data['user']['phone_number'],
                        'message' => $data['message'],
                ));

                $user = User::find($data['user']['id']);

                $user->messages()->create([
                        'to' => $data['user']['phone_number'],
                        'message' => $data['message'],
                ]);

                $job->delete();
        }

}
```

Provavelmente você notou vários problemas ao examinar esta classe. Primeiro, será difícil testá-la. A classe `Twilio_SMS` é instanciada dentro do método `fire`, isso significa que não poderemos injetar um simulador deste serviço. Segundo, estamos usando o Eloquent direto no manipulador, o que exige acessar um banco de dados durante a execução dos testes desta classe. E por último, nós não poderemos enviar mensagens SMS fora do *queue*. Toda a lógica de envio de SMS está fortemente acoplada ao *queue* do Laravel.

Ao extrair esta lógica para uma classe de "serviço" separada, nós estaremos desacoplando a lógica de envio de SMS do *queue* do Laravel. Isso nos permitirá enviar mensagens SMS de qualquer lugar em nossa aplicação. Enquanto desacoplamos este processo do *queue*, vamos aproveitar e torná-lo mais fácil de ser testado.

Vamos examinar uma alternativa:

```
class User extends Eloquent {

        /**
         * Envia uma mensagem SMS p/ o usuário
         *
         * @param  SmsCourierInterface  $courier
         * @param  string  $message
         * @return SmsMessage
         */
        public function sendSmsMessage(SmsCourierInterface $courier, $message)
        {
                $courier->sendMessage($this->phone_number, $message);

                return $this->sms()->create([
                        'to' => $this->phone_number,
                        'message' => $message,
                ]);
        }

}
```

Neste novo exemplo, nós extraímos a lógica de envio de SMS para o modelo `User`. Também injetamos uma implementação da interface `SmsCourierInterface` no método, o que nos permitirá testar melhor este aspecto do processo. Agora, vamos reescrever o manipulador de *queue*:

```
class SendSMS {

        public function __construct(UserRepository $users,
                                    SmsCourierInterface $courier)
        {
                $this->users = $users;
                $this->courier = $courier;
        }

        public function fire($job, $data)
        {
                $user = $this->users->find($data['user']['id']);

                $user->sendSmsMessage($this->courier, $data['message']);

                $job->delete();
        }

}
```

Como você pode ver neste exemplo, nosso manipulador de *queue* ficou "mais leve". Agora ele serve apenas como uma *camada de interpretação* entre o *queue* e a lógica *real* da sua aplicação. Isso é fantástico! Significa que nós podemos facilmente enviar mensagens SMS fora do contexto do manipulador. Para concluir, vamos escrever alguns testes para nossa lógica de envio de SMS:

```
class SmsTest extends PHPUnit_Framework_TestCase {

        public function testUserCanBeSentSmsMessages()
        {
                /**
                 * Preparar...
                 */
                $user = Mockery::mock('User[sms]');
                $relation = Mockery::mock('StdClass');
                $courier = Mockery::mock('SmsCourierInterface');

                $user->shouldReceive('sms')->once()->andReturn($relation);

                $relation->shouldReceive('create')->once()->with(array(
                        'to' => '555-555-5555',
                        'message' => 'Test',
                ));

                $courier->shouldReceive('sendMessage')->once()->with(
                        '555-555-5555', 'Test'
                );

                /**
                 * Ação...
                 */
                $user->sms_number = '555-555-5555';
                $user->sendSmsMessage($courier, 'Test');
        }

}
```

Outros Manipuladores

Nós podemos aperfeiçoar outros tipos de "manipuladores" usando esta mesma técnica. Transformar todos os seus manipuladores em meras *camadas de interpretação* ajudará a manter o "grosso" da sua lógica organizado e desacoplado do resto do *framework*. Para entendermos isso melhor, vamos examinar um filtro de rota que verifica se o usuário atual é assinante do plano "premium".

```
Route::filter('premium', function()
{
        return Auth::user() && Auth::user()->plan == 'premium';
});
```

À primeira vista, este filtro parece inofensivo. O que poderia dar errado em um filtro tão pequeno? Entretanto, mesmo sendo pequeno, ainda faltam detalhes da implementação da nossa aplicação neste código. Observe que nós estamos checando manualmente o valor da variável `plan`. A representação dos "planos" nesta camada estão acopladas à nossa camada de transporte (rota). Se um dia a representação do "plano" *premium* for alterada no banco de dados, ou no modelo do usuário, nós precisaremos reescrever este filtro!

Por isso, vamos fazer uma pequena alteração:

```
Route::filter('premium', function()
{
        return Auth::user() && Auth::user()->isPremium();
});
```

Um mudança pequena como esta traz grandes benefícios a um custo pequeno. Ao permitir que o modelo decida se um usuário é *premium* ou não, nós removemos todos os detalhes da implementação do filtro de rota. O filtro não é mais o responsável em determinar se um usuário está no plano *premium* ou não, agora ele conta com a ajuda do modelo `User`. Assim, se a representação do plano *premium* mudar no banco de dados, não haverá necessidade de atualizar o filtro de rota!

Quem é o responsável?

Estamos mais uma vez explorando o conceito de *responsabilidade*. Lembre-se, considere sempre a responsabilidade de cada classe. Não deixe as camadas de transporte, como os manipuladores, serem responsáveis pela lógica da sua aplicação.

Extendendo o Framework

Introdução

O Laravel oferece muitos pontos de extensão para que você possa personalizar os componentes de seu núcleo, ou mesmo substituí-los completamente. Por exemplo, a classe `Hash` é definido pelo contrato `HasherInterface`, que pode ser implementado de acordo com as necessidades da sua aplicação. Você também pode extender o objeto `Request`, para então adicionar os seus próprios métodos "auxiliares". Você pode até mesmo adicionar novos *drivers* de autenticação, *cache* e sessão!

Geralmente, os componentes do Laravel podem ser extendidos de duas formas: vinculando uma nova implementação ao container IoC ou registrando uma extensão com a classe `Manager`, que é uma implementação do padrão de desenvolvimento *"factory"* (fábrica). Neste capítulo, vamos explorar as várias formas de extender o *framework* e examinar o código necessário para fazê-lo.

Métodos de extensão

Lembre-se, normalmente, os componentes do Laravel são extendidos de duas formas: vinculações IoC e as classes gerenciadoras (*manager*, em inglês). Estas classes servem como uma implementação do padrão de desenvolvimento *factory* e é responsável por instanciar componentes baseados em *drivers*, tais como: *cache* e sessão.

Gerenciadores e Fábricas

O Laravel possui muitas classes gerenciadores que administram a criação de componentes baseados em *drivers*, que incluem: *cache*, sessão, autenticação e *queue*. Esta classe é responsável por criar uma implementação particular do *driver* baseado na configuração da aplicação. Por exemplo, a classe `CacheManager` pode criar uma implementação do *driver* de cache APC, Memcached, *Native* e muitas outras.

Cada um desses "gerenciadores" possui um método chamado `extend`, que facilita a injeção da resolução de novas funcionalidades dos *drivers* no gerenciador. A seguir, nós cobriremos cada um desses gerenciadores vendo exemplos de como podemos injetar suporte para um *driver* personalizado neles.

Aprenda sobre os seus gerenciadores

Separe um tempo para explorar as classes gerenciadoras inclusas no Laravel, como as `CacheManager` e `SessionManager`. Ler estas classes lhe dará um conhecimento mais profundo de como o *framework* Laravel funciona. Todas elas extendem a classe base `Illuminate\Support\Manager`, que provê funcionalidades comuns muito úteis aos gerenciadores.

Cache

Para extender o *cache* usado no Laravel, usaremos o método `extend` no `CacheManager`, que serve para vincular ao gerenciador um resolvedor de *driver* personalizado. Este método está presente em todas as classes gerenciadoras. Por exemplo, para registrar um novo *driver* de cache chamado "mongo", faremos o seguinte:

```
Cache::extend('mongo', function($app)
{
        // Retornar uma instância de Illuminate\Cache\Repository...
});
```

O primeiro argumento passado ao método `extend` é o nome do novo *driver*. Este é o nome que será usado na opção `driver` no arquivo de configuração `app/config/cache.php`. Já o segundo argumento, é uma *closure* que retorna uma instância de `Illuminate\Cache\Repository`. A *closure* receberá uma instância de `$app`, que por sua vez é uma instância de `Illuminate\Foundation\Application` e também é um container IoC.

Para criar o nosso *driver* de cache personalizado, nós primeiro precisamos implementar o contrato `Illuminate\Cache\StoreInterface`. Nossa implementação de cache para o MongoDB deverá ficar assim:

```
class MongoStore implements Illuminate\Cache\StoreInterface {

        public function get($key) {}
        public function put($key, $value, $minutes) {}
        public function increment($key, $value = 1) {}
        public function decrement($key, $value = 1) {}
        public function forever($key, $value) {}
        public function forget($key) {}
        public function flush() {}

}
```

Agora, só precisamos implementar estes métodos usando uma conexão com o MongoDB. Quando esta implementação ficar pronta, poderemos finalizar o registro do nosso *driver* personalizado:

```
use Illuminate\Cache\Repository;

Cache::extend('mongo', function($app)
{
        return new Repository(new MongoStore);
});
```

Como você pode ver acima, é possível usar a classe base `Illuminate\Cache\Repository` na criação de *drivers* de cache personalizados. Normalmente, você não precisará criar a sua própria classe repositória.

Se você tem dúvida sobre onde este código deve ir, considere disponibilizá-lo no Packgist[7]! Ou então, crie um *namespace* chamado `Extensions` na pasta principal da sua aplicação. Por exemplo, se sua aplicação se chama `Snappy`, seu código pode ser colocado em `app/Snappy/Extensions/MongoStore.php`. Entretanto, tenha em mente que o Laravel não é rígido quanto à estrutura da sua aplicação, você é livre para organizá-la conforme suas preferências.

Onde extender

Sempre que você estiver em dúvida sobre onde colocar um código, considere usar um provedor de serviços. Conforme já discutimos, eles são ótimos para organizar o código das suas extensões do *framework*.

Sessão

Extender o *driver* de sessão do Laravel é tão fácil quanto extender seu sistema de *cache*. Mais uma vez, usaremos o método `extend` para registrar nosso código:

```
1  Session::extend('mongo', function($app)
2  {
3          // Retorna uma implementação de SessionHandlerInterface
4  });
```

Note que o nosso *driver* de *cache* deve implementar o contrato `SessionHandlerInterface`. Esta interface faz parte do núcleo do PHP 5.4+. Se você está usando a versão 5.3 do PHP, esta interface será definida automaticamente pelo Laravel. Ela contem apenas alguns métodos que nós precisamos implementar. Nossa implementação para o MongoDB ficará mais ou menos assim:

```
1  class MongoHandler implements SessionHandlerInterface {
2
3          public function open($savePath, $sessionName) {}
4          public function close() {}
5          public function read($sessionId) {}
6          public function write($sessionId, $data) {}
7          public function destroy($sessionId) {}
8          public function gc($lifetime) {}
9
10 }
```

Como estes métodos não são tão claros quanto os da `StoreInterface` do *cache*, veremos rapidamente o que cada um deles faz:

[7] https://packagist.org

- O método `open` normalmente é usado em sistemas que armazenam sessões em arquivos. Como o Laravel já vem como um *driver* de sessão `native` que utiliza o armazenamento em arquivo nativo do PHP, você quase nunca precisará colocar algo neste método. Deixe-o em branco. Este é apenas um caso de mal planejamento de interface (que discutiremos a seguir) por parte do PHP.
- O método `close`, assim como o método `open`, na maioria dos casos poderá ser desconsiderado.
- O método `read` deve retornar uma *string* dos dados da sessão associada com a `$sessionId` fornecida. Os dados das sessões não precisam ser serializados nem codificados pelo *driver* quando estes forem lidos ou gravados; o Laravel fará isso por você.
- O método `write` deve gravar os dados da *string* `$data` associados com a `$sessionId` em algum tipo de sistema de armazenamento persistente, como o MongoDB, Dynamo e etc.
- O método `destroy` deve remover os dados associados com a `$sessionId` do armazenamento persistente.
- O método `gc` deve destruir os dados de todas as sessões anteriores à data `$lifetime`, que é uma UNIX *timestamp*. Para os sistemas que expiram estes dados automaticamente, como Memcached e Redis, este método deve ser deixado em branco.

Assim que a implementação do contrato `SessionHandlerInterface` estiver pronta, poderemos registrá-la com o gerenciador de sessão:

```
1  Session::extend('mongo', function($app)
2  {
3          return new MongoHandler;
4  });
```

Feito o registro, o novo *driver* de sessão `mongo` estará pronto para ser usado no arquivo de configuração `app/config/session.php`.

Compartilhe seu conhecimento

Lembre-se, se um dia você escrever um *driver* de sessão personalizado, compartilhe-o no Packagist!

Autenticação

A autenticação pode ser extendida da mesma forma que os componentes *cache* e sessão. Por isso, continuaremos usando o método `extend` que já estamos acostumados:

```
Auth::extend('riak', function($app)
{
        // Retorna uma implementação de
        // Illuminate\Auth\UserProviderInterface
});
```

As implementações da `UserProviderInterface` são responsáveis apenas por obter uma implementação da `UserInterface` de um sistema de armazenamento persistente, como MySQL, Riak e etc. Estas duas interfaces permitem que o mecanismo de autenticação do Laravel funcione independente de onde os dados do usuário estão armazenados ou de qual tipo de classe é usada para representá-lo.

Vamos dar uma espiada na `UserProviderInterface`:

```
interface UserProviderInterface {

        public function retrieveById($identifier);
        public function retrieveByCredentials(array $credentials);
        public function validateCredentials(UserInterface $user, array $credentials);

}
```

A função `retrieveById` tipicamente recebe uma chave numérica que representa o usuário, como uma ID auto-incrementada do banco de dados MySQL. Este método deve obter e retornar uma implementação do contrato `UserInterface` correspondente à ID fornecida.

O método `retrieveByCredentials` recebe o *array* de credenciais passadas ao método `Auth::attempt` durante a tentativa de entrar em sua aplicação. Em seguida, este método deve "buscar" um usuário correspondente a tais credenciais no armazenamento persistente. Geralmente, este método executará uma *query* com uma condição *where* em `$credentails['username']`. **Este método jamais deve tentar validar uma senha ou autenticar um usuário**.

O método `validateCredentials` deve comparar o `$user` fornecido com as `$credentials` para autenticá-lo. Por exemplo, este método poderia comparar a *string* `$user->getAuthPassword()` com uma `Hash::make` de `$credentials['password']`.

Agora que já exploramos todos os métodos da `UserProviderInterface`, vamos dar uma olhada na `UserInterface`. Lembre-se, o provedor deve retornar implementações desta interface a partir dos métodos `retrieveById` e `retrieveByCredentials`.

```
interface UserInterface {

        public function getAuthIdentifier();
        public function getAuthPassword();

}
```

Esta interface é simples. O método `getAuthIdentifier` deve retornar a "chave primária" do usuário. No caso do MySQL, esta seria a *primary key* auto-incrementada. Já o método `getAuthPassword` deve retornar o *hash* da senha do usuário. Esta interface permite que o sistema de autenticação funcione com qualquer classe `User`, independente do ORM ou sistema de armazenamento escolhido. Por padrão, o Laravel inclui uma classe `User` que implementa esta interface na pasta `app/models`; use-a como um exemplo de implementação.

Para finalizar, quando implementarmos o contrato `UserProviderInterface`, nossa extensão estará pronta para ser registrada com a *façade* `Auth`:

```
1  Auth::extend('riak', function($app)
2  {
3          return new RiakUserProvider($app['riak.connection']);
4  });
```

Tendo registrado o *driver* com o método `extend`, você poderá escolhê-lo no seu arquivo de configuração `app/config/auth.php` usando a chave `riak`.

Extensões Baseadas no Container IoC

Quase todos os provedores de serviços inclusos no Laravel vinculam objetos ao container IoC. A lista dos provedores de serviços da sua aplicação pode ser encontrada no arquivo `app/config/app.php`. Tendo um tempo, seria bom que você analisasse o código-fonte de cada provedor nesta lista. Fazer isso lhe ajudará a entender melhor o que cada provedor acrescenta ao *framework* e quais "chaves" (nomes) são usadas para vincular vários serviços ao container IoC.

Por exemplo, o `PaginationServiceProvider` vincula a chave `paginator` ao container IoC, que resolve em uma instância de `Illuminate\Pagination\Environment`. Substituindo esta vinculação, você poderá extender ou substituir esta classe facilmente. Por exemplo, vamos criar uma classe que extende a classe base `Environment`:

```
1  namespace Snappy\Extensions\Pagination;
2
3  class Environment extends \Illuminate\Pagination\Environment {
4
5          //
6
7  }
```

Agora, você já pode criar um novo provedor de serviço `SnappyPaginationProvider` para subtituir o paginador do Laravel usando o método `boot`:

```
class SnappyPaginationProvider extends PaginationServiceProvider {

        public function boot()
        {
                App::bind('paginator', function()
                {
                        return new Snappy\Extensions\Pagination\Environment;
                });

                parent::boot();
        }

}
```

Note que esta classe extende o provedor `PaginationServiceProvider` e não a classe base padrão `ServiceProvider`. Feito isto, substitua o provedor de serviço `PaginationServiceProvider` por sua nova extensão no arquivo de configuração `app/config/app.php`.

Esta é a forma geral de extender qualquer classe do núcleo que está vinculada ao container. Todas as classes do núcleo estão vinculadas ao container desta forma e podem ser facilmente substituídas. Novamente, leia o código dos provedores de serviço inclusos no *framework* para aprender onde as várias classes são vinculadas ao container e quais chaves são usadas. Esta é uma excelente forma de aprender como as várias partes do Laravel são unidas.

Request

A classe `Request` é uma peça fundamental do *framework* e é instanciada muito cedo no ciclo da sua aplicação. Por isso, ela não é extendida como os exemplos anteriores.

Primeiro, extenda a classe normalmente:

```
<?php namespace QuickBill\Extensions;

class Request extends \Illuminate\Http\Request {

        // Personalização e métodos auxiliares vão aqui...

}
```

Agora, abra o arquivo `bootstrap/start.php`. Este é um dos primeiros arquivos incluídos na sua aplicação. Note que a primeira ação neste arquivo é a criação de uma instância da sua aplicação:

```
$app = new \Illuminate\Foundation\Application;
```

Quando esta nova instância `$app` for criada, uma nova instância de `Illuminate\Http\Request` também será criada e vinculada ao container IoC usando a chave `request`. Assim, nós precisamos encontrar uma forma de especificar qual classe deve ser usada como tipo de *request* "padrão", certo? Felizmente, o método `requestClass` na instância da sua aplicação faz exatamente isso! Só precisamos acrescentar esta linha no começo do arquivo `bootstrap/start.php`:

```
1  use Illuminate\Foundation\Application;
2
3  Application::requestClass('QuickBill\Extensions\Request');
```

Agora, o Laravel sempre usará a classe personalizada que você especificou para criar uma instância de `Request`. Assim, você sempre terá uma instância dessa classe à sua disposição, até mesmo nos testes de unidade!

Princípio da Responsabilidade Única

Introdução

Os princípios de desenvolvimento "SOLID", articulados por Robert "Uncle Bob" Martin, são cinco princípios que fornecem um alicerce firme para o desenvolvimento de aplicações. São eles:

- Princípio da Responsabilidade Única
- Princípio do Aberto/Fechado
- Princípio de Substituição de Liskov
- Princípio da Segregação de Interface
- Princípio da Inversão de Dependência

Analisaremos a fundo todos estes princípios e veremos alguns exemplos de código ilustrando cada um deles. Logo você descobrirá que cada princípio completa os demais. Assim, se um deles falhar, a maioria ou todos os demais também falharão.

Em Ação

O Princípio da Responsabilidade Única diz que uma classe deve mudar por um, e apenas um, motivo. Em outras palavras, o escopo e responsabilidade da classe devem ser estritamente focados. Como eu já disse, ignorância é uma bênção em se tratando das responsabilidades de uma classe. Ela deve executar o seu trabalho sem ser afetada pelas mudanças em suas dependências.

Considere a seguinte classe:

```php
class OrderProcessor { // Processador de pedidos

  public function __construct(BillerInterface $biller)
  {
    $this->biller = $biller;
  }

  public function process(Order $order)
  {
    $recent = $this->getRecentOrderCount($order);

    if ($recent > 0)
    {
      throw new Exception('Pedido provavelmente duplicado.');
    }

    $this->biller->bill($order->account->id, $order->amount);

    DB::table('orders')->insert(array(
      'account'    => $order->account->id,
      'amount'     => $order->amount;
      'created_at' => Carbon::now();
    ));
  }

  protected function getRecentOrderCount(Order $order)
  {
    $timestamp = Carbon::now()->subMinutes(5);

    return DB::table('orders')
                  ->where('account', $order->account->id)
                  ->where('created_at', '>=', $timestamps)
                  ->count();
  }

}
```

Quais são as responsabilidades da classe acima? Obviamente, seu nome implica que sua responsabilidade é processar pedidos. Porém, analisando o método `getRecentOrderCount`, percebemos que ele examina o histórico de pedidos de uma conta no banco de dados para detectar se há pedidos duplicados. Com estas responsabilidades extras, quando o armazenamento de dados ou as regras de validação mudarem, nós precisaremos alterar esta classe também.

Estas responsabilidades extras devem ser extraídas para uma outra classe, pode ser a classe `OrderRepository`:

```php
class OrderRepository { // Repositório de pedidos

  public function getRecentOrderCount(Account $account)
  {
    $timestamp = Carbon::now()->subMinutes(5);

    return DB::table('orders')
                    ->where('account', $account->id)
                    ->where('created_at', '>=', $timestamp)
                    ->count();
  }

  public function logOrder(Order $order)
  {
    DB::table('orders')->insert(array(
      'account'    => $order->account->id,
      'amount'     => $order->amount;
      'created_at' => Carbon::now();
    ));
  }

}
```

Agora, nós podemos injetar nosso repositório no `OrderProcessor`, aliviando assim a sua responsabilidade de analisar o histórico de pedidos:

```php
class OrderProcessor { // Processador de pedidos

  public function __construct(BillerInterface $biller,
                              OrderRepository $orders)
  {
    $this->biller = $biller;
    $this->orders = $orders;
  }

  public function process(Order $order)
  {
    $recent = $this->orders->getRecentOrderCount($order->account);

    if ($recent > 0)
    {
      throw new Exception('Pedido provavelmente duplicado.');
    }

    $this->biller->bill($order->account->id, $order->amount);

    $this->orders->logOrder($order);
  }

}
```

Tendo abstraído a responsabilidade de reunir os dados dos pedidos, não precisaremos mais alterar nossa classe `OrderProcessor` quando os métodos para obter e registrar pedidos mudarem. As responsabilidades da nossa classe foram focadas e definidas, tornando nossa aplicação mais fácil de manter e seu código mais limpo e expressivo.

Tenha em mente que, aplicar o Princípio da Responsabilidade Única não é apenas escrever menos linhas de código. É escrever classes que possuem uma responsabilidade estrita e coesa com os métodos nela contidos. Certifique-se de que todos os métodos estão alinhados com a responsabilidade da classe. Após criarmos uma biblioteca de classes limpas e pequenas, cada uma tendo responsabilidades bem definidas, nosso código estará desacoplado e poderá ser testado e modificado com mais facilidade.

Princípio do Aberto/Fechado

Introdução

Durante a vida de uma aplicação, mais tempo é gasto acrescentando ao código existente do que adicionando novos recursos escritos do zero. E como você já deve ter notado, isso pode ser um processo entediante. Sempre que o código é modificado, você corre o risco de introduzir novos *bugs* ou quebrar funcionalidades completas. O ideal seria poder modificar um projeto tão rápido e fácil quanto começar do zero. Isso é possível se desenvolvermos nossa aplicação de acordo como o princípio do aberto/fechado!

Definição

O Princípio do Aberto/Fechado diz que o código deve estar aberto para extensão, mas fechado para modificação.

Em Ação

Para demonstrar este princípio, continuaremos trabalhando com a classe `OrderProcessor` do capítulo anterior. Considere o método `process` a seguir:

```
1  $recent = $this->orders->getRecentOrderCount($order->account);
2
3  if ($recent > 0)
4  {
5    throw new Exception('Pedido provavelmente duplicado.');
6  }
```

Este código é bem legível e testável, pois estamos injetando a dependência. Entretanto, e se as regras da validação dos pedidos mudarem? E se a nossa aplicação crescer a ponto de termos *muitas* regras novas? O método `process` crescerá rapidamente e se tornará um monstro de código espaguete quase impossível de ser mantido. Tudo isso porque este código precisará ser modificado sempre que uma nova regra de validação for adicionada. Ou seja, ele viola o princípio do aberto/fechado, pois está aberto para modificação. Lembre-se, nosso queremos que o nosso código esteja aberto para *extensão*, não para modificação.

Ao invés de validar os pedidos no método `process`, vamos definir uma nova interface `OrderValidator`:

```php
interface OrderValidatorInterface {
  public function validate(Order $order);
}
```

Em seguida, vamos definir uma implementação para previnir a duplicidade de pedidos:

```php
class RecentOrderValidator implements OrderValidatorInterface {

  public function __construct(OrderRepository $orders)
  {
    $this->orders = $orders;
  }

  public function validate(Order $order)
  {
    $recent = $this->orders->getRecentOrderCount($order->account);

    if ($recent > 0)
    {
      throw new Exception('Pedido provavelmente duplicado.');
    }
  }

}
```

Ótimo! Agora nós temos um método pequeno, testável e encapsulado. Vamos escrever outra implentação para verificar se a conta está suspensa:

```php
class SuspendedAccountValidator implememts OrderValidatorInterface {

  public function validate(Order $order)
  {
    if ($order->account->isSuspended())
    {
      throw new Exception("Contas suspensas não podem realizar pedidos.")
    }
  }

}
```

Agora que temos duas implementações diferentes da nossa `OrderValidatorInterface`, vamos usá-la na classe `OrderProcessor`. Vamos injetar um *array* de validadores na instância do processador, assim poderemos acrescentar ou remover regras de validação conforme o nosso código for crescendo.

```
class OrderProcessor {

  public function __construct(BillerInterface $biller,
                              OrderRepository $orders,
                              array $validators = array())
  {
    $this->biller = $biller;
    $this->orders = $orders;
    $this->validators = $validators;
  }

}
```

Em seguida, usando um *loop*, poderemos validar todas as regras dentro do método `process`:

```
public function process(Order $order)
{
  foreach ($this->validators as $validator)
  {
    $validator->validate($order);
  }

  // Processar pedido válido...
}
```

Finalizando, vamos registrar a classe `OrderProcessor` no container IoC da aplicação:

```
App::bind('OrderProcessor', function()
{
  return new OrderProcessor(
    App::make('BillerInterface'),
    App::make('OrderRepository'),
    array(
      App::make('RecentOrderValidator'),
      App::make('SuspendedAccountValidator'),
    ),
  );
});
```

Com estas pequenas modificações, agora nós podemos acrescentar e remover novas regras de validação sem mudar uma única linha do código existente. Cada regra nova será simplesmente uma implementação da interface `OrderValidatorInterface` e será registrada no container IoC. Ao invés de testarmos um método `process` gigantesco, poderemos testar cada regra de validação isoladamente. Nosso código está *aberto* para extensão, mas *fechado* para modificação.

Vazamentos em abstrações

Fique de olho nas dependências que vazam detalhes da implementação. Uma mudança de implementação numa dependência não deve exigir mudanças em seus consumidores. Quando é necessário mudar o consumidor, nós dizemos que a dependência está "vazando" detalhes da implementação. Se isso acontecer em suas abstrações, provavelmente você está violando o Princípio do Aberto/Fechado.

Antes de avançarmos, lembre-se que estes princípios não são leis. Não é obrigatório que cada parte da sua aplicação seja "plugável". Uma aplicação pequena que acessa um banco de dados MySQL para obter alguns registros não precisa aplicar todos os princípios de desenvolvimentos imagináveis. Não aplique estes princípios cegamente apenas para desencargo de consciência, pois em fazê-lo você estaria planejado excessivamente e criando um sistema complicado. Tenha em mente que estes princípios foram criados para resolverem problemas estruturais comuns em aplicações grandes e robustas. Tendo dito isto, não use este parágrafo como uma desculpa para sua preguiça!

Princípio da Substituição de Liskov

Introdução

Não se preocupe, o Princípio da Substituição de Liskov é mais fácil de entender do que parece. Ele afirma que você deve poder usar qualquer implementação de uma abstração onde quer que a abstração seja aceita. Simplificando, este princípio diz que: se uma classe usa a implementação de uma interface, ele deve aceitar qualquer outra implementação dessa interface sem precisar de qualquer modificação.

Definição

Objetos devem ser substituíveis por instâncias dos seus subtipos sem alterar o funcionamento do programa.

Em Ação

Para ilustrar este princípio, continuaremos usando como exemplo a classe `OrderProcessor` do capítulo anterior. Dê uma olhada neste método:

```
1  public function process(Order $order)
2  {
3    // Validar pedido...
4
5    $this->orders->logOrder($order);
6  }
```

Observe que após a validação do pedido, nós registramos o pedido usando a implementação `OrderRepositoryInterface`. Vamos supor que no princípio, todos os nossos pedidos eram processados e armazenados no formato CSV em um arquivo. A única implementação de `OrderRepositoryInterface` era `CsvOrderRepository`. Agora, como o número de pedidos cresceu, queremos usar um banco de dados relacional para armazená-los. Assim, vamos analisar uma possível implementação para o nosso novo repositório:

```php
class DatabaseOrderRepository implements OrderRepositoryInterface {

  protected $connection;

  public function connect($username, $password)
  {
    $this->connection = new DatabaseConnection($username, $password);
  }

  public function logOrder(Order $order)
  {
    $this->connection->run('insert into orders values (?, ?)', array(
      $order->id, $order->amount,
    ));
  }

}
```

Agora, vamos examinar como podemos usar esta implementação:

```php
public function process(Order $order)
{
        // Validar pedido...

        if ($this->repository instanceof DatabaseOrderRepository)
        {
                $this->repository->connect('root', 'password');
        }

        $this->respository->logOrder($order);
}
```

Note que fomos obrigados a verificar que a `OrderRepositoryInterface` é uma implementação de um banco de dados na classe que processa os pedidos. E em sendo, a conexão é realizada. Isso não será um problema em aplicações menores, mas imagine se a interface `OrderRepositoryInterface` for usada em várias classes? Esse mesmo código seria escrito repetidas vezes e mantê-lo nos daria uma grande dor de cabeça, além de poder criar *bugs*. E se esquecermos de atualizá-lo em uma única classe, nossa aplicação poderá quebrar completamente.

O exemplo acima claramente não aplica o Princípio da Substituição de Liskov, pois não pudemos injetar uma implementação da nossa interface sem mudar também a classe que invoca o método `connect`. Assim, tendo identificado o problema, agora nós vamos solucioná-lo. Veja a nossa nova implementação `DatabaseOrderRepository`

```php
class DatabaseOrderRepository implements OrderRepositoryInterface {

        protected $connector;

        public function __construct(DatabaseConnector $connector)
        {
                $this->connector = $connector;
        }

        public function connect()
        {
                return $this->connector->bootConnection();
        }

        public function logOrder(Order $order)
        {
                $connection = $this->connect();

                $connection->run('insert into orders values (?, ?)', array(
                        $order->id, $order->amount,
                ));
        }

}
```

Agora, nós podemos remover nosso código de inicialização da classe consumidora e o repositório `DatabaseOrderRepository` gerenciará a conexão com o banco de dados:

```php
public function process(Order $order)
{
        // Validar pedido...

        $this->respository->logOrder($order);
}
```

Com esta modificação, agora nós podemos usar os repositórios `CsvOrderRepository` ou `DatabaseOrderReposito` sem modificar a classe consumidora `OrderProcessor`. Nosso código está de acordo com o Princípio da Substituição de Liskov! Você notou que muitos dos conceitos de arquitetura que discutimos estão relacionados com o *conhecimento*? Especificamente, o conhecimento que uma classe possui dos seus "arredores", tais como códigos periféricos e dependências que ajudam uma classe em sua tarefa. Na sua jornada rumo a uma aplicação com arquitetura robusta, limitar o *conhecimento* de uma classe será um tema importante e recorrente.

Note também a consequência da violação deste princípio no que diz respeito ao demais princípios já vistos. Ao "quebrar" este princípio, o Princípio do Aberto/Fechado também é violado, pois ao

verificar as instâncias de várias classes filhas, a classe consumidora precisará ser alterada sempre que houver uma nova classe filha.

Cuidado com os vazamentos

Você deve ter notado que este princípios está intimamente relacionado com a prevenção dos "vazamentos de abstrações" discutidos no capítulo anterior. O vazamento da abstração no repositório do banco de dados foi a primeira indicação de que o Princípio da Substituição de Liskov estava sendo violado. Fique atento com estes vazamentos!

Princípio da Segregação de Interface

Introdução

O Princípio da Segregação de Interface afirma que nenhuma implementação de uma interface deve ser forçada a depender dos métodos não usados por ela. Você já precisou implementar métodos de uma interface que você acabou não usando? Se sim, você provavelmente criou métodos em branco em sua implementação. Este é um exemplo de ser forçado a usar uma interface que viola o princípio deste capítulo.

Em termos práticos, este princípio exige que as interfaces sejam granulares e focadas. Soa familiar? Lembre-se, todos os cinco princípios SOLID estão tão relacionados que, ao violar um, geralmente você estará violando os demais também. Para violar o Princípio da Segregação de Interface, você também precisa violar o Princípio da Responsabilidade Única.

Ao invés de uma interface "gorda" contendo métodos que não serão usados por todas as implementações, é melhor termos várias interfaces menores que podem ser implementadas individualmente conforme necessário. Tendo contratos focados e menores, o código consumidor poderá depender em interfaces menores sem depender das partes da aplicação que elas não usam.

Definição

Este princípio afirma que nenhuma implementação de uma interface deve ser forçada a depender dos métodos não usados por ela.

Em Ação

Para ilustrar este princípio, vamos considerar uma biblioteca para manipular sessões. Na verdade, vamos usar a interface `SessionHandlerInterface` do próprio PHP. Estes são os métodos definidos por esta interface, inclusos no PHP na versão 5.4:

```
1  interface SessionHandlerInterface {
2          public function close();
3          public function destroy($sessionId);
4          public function gc($maxLiftetime);
5          public function open($savePath, $name);
6          public function read($sessionId);
7          public function write($sessionId, $sessionData);
8  }
```

Agora que você está familiarizado com os métodos desta interface, considere uma implementação usando Memcached. A implementação Memcached desta interface precisará definir funcionalidades para cada um destes métodos? Não! Na realidade, não precisaremos implementar nem metade deles!

Como o Memcached expirará automaticamente os valores armazenados nele, não precisaremos implementar o método `gc` nesta interface, bem como os métodos `open` e `close`. Assim, somos forçados a definir métodos em branco para estes métodos em nossa implementação. Para corrigir este problema, vamos começar definindo uma interface menor e mais focada para coletar o lixo (*garbage collector* ou GC, em inglês) das sessões:

```
interface GarbageCollectorInterface {
        public function gc($maxLifetime);
}
```

Com uma interface menor, qualquer código consumidor poderá depender neste contrato conciso, pois ele define um conjunto de funções pequeno e não cria uma dependência completa no manipulador de sessões.

Para compreendermos este princípio melhor, vamos consolidar nosso conhecimento com outro exemplo. Imagine uma classe Eloquent `Contact` definida desta forma:

```
class Contact extends Eloquent {

        public function getNameAttribute()
        {
                return $this->attributes['name'];
        }

        public function getEmailAttribute()
        {
                return $this->attributes['email'];
        }

}
```

Agora, vamos assumir que nossa aplicação também emprega uma classe `PasswordReminder` que é responsável pelo envio de lembretes de senha por e-mail. Abaixo temos uma possível definição para esta classe:

```
class PasswordReminder {

        public function remind(Contact $contact, $view)
        {
                // Enviar o lembrete de senha por e-mail...
        }

}
```

Como você deve ter notado, nossa classe `PasswordReminder` depende da classe `Contact`, que por sua vez depende do ORM Eloquent. Não é desejável nem foi necessário acoplar o sistema de lembrete de senhas a uma implementação específica do ORM Eloquent. Ao quebrar esta dependência, nós podemos alterar livremente o mecanismo de armazenamento no *back-end* ou o ORM sem afetar este componente da nossa aplicação. Novamente, ao violarmos um dos princípios SOLID, estamos dando à classe consumidora muito *conhecimento* sobre o restante da aplicação.

Para quebrar esta dependência, vamos criar a interface `RemindableInterface`. Na verdade, tal interface está inclusa no Laravel e por padrão é implementada pelo modelo `User`:

```
interface RemindableInterface {
  public function getReminderEmail();
}
```

Quando a interface estiver pronta, poderemos implementá-la no nosso modelo:

```
class Contact extends Eloquent implements RemindableInterface {

  public function getReminderEmail()
  {
    return $this->email;
  }

}
```

Para finalizar, agora nós podemos depender desta interface na classe `PasswordReminder`:

```
class PasswordReminder {

  public function remind(RemindableInterface $remindable, $view)
  {
    // Enviar o lembrete de senha por e-mail...
  }

}
```

Fazendo esta pequena alteração, nós removemos qualquer dependência desnecessária do componente de lembrete de senhas e ele tornou-se flexível o bastante para usar qualquer classe de qualquer ORM, desde que tal classe implemente a nova `RemindableInterface`. É exatamente assim que o componente de lembrete de senhas do Laravel aceita qualquer banco de dados e ORM!

Saber é poder

Mais uma vez, discutimos as armadilhas de dar à classe muito conhecimento sobre os detalhes da implementação da aplicação. Se prestarmos bastante cuidado com a quantidade de conhecimento passado para uma classe, conseguiremos aplicar todos os princípios SOLID.

Princípio da Inversão de Dependência

Introdução

Chegamos ao destino final de nosso estudo sobre os cinco princípios de desenvolvimento SOLID! O último princípio é o Princípio da Inversão de Dependência; ele afirma que: o código de alto nível não deve depender do código de baixo nível. Pelo contrário, um código de alto nível deve depender de uma camada de abstração, que funciona como um mediador entre o código de alto e o de baixo nível. Um segundo aspecto deste princípio é que as abstrações não devem depender dos detalhes, mas os detalhes devem depender das abstrações. Se tudo isso parece confuso agora, não se preocupe, veremos sobre ambos os aspectos logo a seguir.

Definição

Este princípio afirma que o código de alto nível não deve depender do código de baixo nível e que as abstrações não devem depender dos detalhes.

Em Ação

Se você já leu os capítulos anteriores deste livro, você já compreendeu o Princípio da Inversão de Dependência! Para ilustrá-lo, vamos considerar a seguinte classe:

```
class Authenticator {

        public function __construct(DatabaseConnection $db)
        {
                $this->db = $db;
        }

        public function findUser($id)
        {
                return $this->db->exec('select * from users where id = ?', array($id));
        }

        public function authenticate($credentials)
        {
                // Autenticação do usuário...
        }

}
```

Como você deve ter adivinhado, a classe `Authenticator` é responsável por encontrar e autenticar nossos usuários. Vamos examinar o construtor desta classe. Veja que estamos induzindo o tipo de uma instância do `DatabaseConnection`. Fazendo assim, estamos acoplando o autenticador ao banco de dados, em outras palavras, estamos dizendo que os usários sempre estarão registrados em um banco de dados SQL. Além do mais, o código de alto nível (o `Authenticator`) diretamente depende do código de baixo nível (o `DatabaseConnection`).

Em primeiro lugar, vamos discutir código de "alto" e de "baixo" nível. O código de baixo nível implementa operações básicas como: ler arquivos no disco ou interagir com o banco de dados. Já o código de alto nível, encapsula lógicas complexas e precisa do código de baixo nível para poder funcionar, mas não devem estar diretamente acoplado um ao outro. Pelo contrário, o código de alto nível deve depender de uma abstração entre ele e o código de baixo nível, como uma interface. E não apenas isso, o código de baixo nível *também* deve depender de uma abstração. Por isso, vamos escrever uma interface para usarmos dentro do nosso `Authenticator`:

```
interface UserProviderInterface {
        public function find($id);
        public function findByUsername($username);
}
```

Agora, vamos injetar uma implementação desta interface na classe `Authenticator`:

```
class Authenticator {

        public function __construct(UserProviderInterface $users,
                                    HasherInterface $hash)
        {
                $this->hash = $hash;
                $this->users = $users;
        }

        public function findUser($id)
        {
                return $this->users->find($id);
        }

        public function authenticate($credentials)
        {
                $user = $this->users->findByUsername($credentials['username']);

                return $this->hash->make($credentials['password']) == $user->password;
        }

}
```

Após estas mudanças, nosso `Authenticator` passou depender de duas abstrações: `UserProviderInterface` e `HasherInterface`. Estamos livres para injetar qualquer implementação destas interfaces no `Authenticator`. Por exemplo, se os nossos usuários forem armazenados no Redis, poderemos escrever um provedor `RedisUserProvider` que implementa o contrato `UserProvider`. Agora, o nosso `Authenticator` não depende diretamento no sistema de armazenamento de baixo nível.

E ainda mais, o nosso código de baixo nível agora depende da abstração de alto nível `UserProviderInterface`, pois ele implementa a interface também:

```php
class RedisUserProvider implements UserProviderInterface {

        public function __construct(RedisConnection $redis)
        {
                $this->redis = $redis;
        }

        public function find($id)
        {
                $this->redis->get('users:'.$id);
        }

        public function findByUsername($username)
        {
                $id = $this->redis->get('user:id:'.$username);

                return $this->find($id);
        }

}
```

Pensamento invertido

Aplicar este princípio *inverte* o modo como muitos desenvolvedores desenvolvem suas aplicações. Ao invés de acoplar o código de alto nível diretamente a um código de baixo nível, no estilo de "cima para baixo", este princípio afirma que **ambos** os códigos de alto e de baixo nível devem depender de uma abstração de alto nível.

Antes de "invertermos" as dependências do nosso `Authenticator`, ele só poderia ser usado com um banco de dados relacional. Se mudássemos algo no sistema de armazenamento, o `Authenticator` precisaria ser modificado também, violando assim o Princípio do Aberto/Fechado. Novamente, múltiplos princípios "andam" lado-a-lado e "caem" juntos.

Ao forçarmos o `Authenticator` a depender de uma abstração e não da camada de armazenamento, ela passou a aceitar qualquer sistema de armazenamento que implemente o contrato `UserProviderInterface` sem precisarmos modificar a classe `Authenticator`. A corrente convencional de dependência foi *invertida* e o nosso código tornou-se muito mais flexível e acolhedor de mudanças!