



# 7

**técnicas  
masters  
para você  
decolar com  
Laravel**



**10 BÔNUS LARAVEL ELOQUENT HACKS**  
que você provavelmente não sabe!

# Introdução

O Laravel é um dos frameworks mais consagrados dos últimos tempos e, logicamente, bastante amado pela maioria dos programadores PHP.

Essa ferramenta tem como principal objetivo permitir que os desenvolvedores de softwares trabalhem de forma rápida e organizada, produzindo um código elegante, limpo e ao mesmo tempo funcional.

Desde 2015 utilizo essa ferramenta como ponto de partida em meus projetos e digo por experiência própria: o Laravel facilita muito nosso trabalho. Durante esse tempo, aprendi técnicas incríveis que me ajudaram (e continuam ajudando) a “codar” melhor os projetos, utilizando esse framework. Tenho certeza que você ainda não conhece muitas delas.

**Então pensei, por que não criar um e-book e compartilhar todo esse conhecimento com outros programadores? E assim nasceu esse material.**

Espero que aproveitem e façam bom uso dessas 7 técnicas masters que eu separei e mais 10 dicas bônus sobre o Laravel Eloquent e que todas essas dicas te ajudem, assim como me ajudaram!

**Então vamos combinar o seguinte:** se ao final da leitura você achar que este e-book te entregou algum conteúdo de valor, você assume o compromisso de divulgá-lo para amigos programadores para que eles também possam beneficiar-se do mesmo conteúdo que você se beneficiou, ok?

Vamos compartilhar conhecimento! 🙌 Ficarei muito grato! ✈️

**E para compartilhar, basta enviar esta URL para seus amigos:**

🔗 <https://pro.tiagomatos.com/curso-laravel-pro>

Aproveite a leitura e vamos decolar. 🚀🚀🚀

# Quem sou eu?

**Antes de começar, permita me apresentar.**

Me chamo Tiago Matos e iniciei minha carreira profissional aos 16 anos...e lá se vão mais de 20 anos de experiência!

Ao longo desses anos, trabalhei bastante na área de desenvolvimento web, o que me fez adquirir muito conhecimento e aprender vários truques.

Fui sócio e líder de projeto na minha agência digital por 9 anos e, desde 2015, atuo como engenheiro de software no mercado australiano.



The screenshot shows a YouTube channel page for "Tiago Matos" with 5,35 mil inscritos. The channel has an "INSCRITO" button with a bell icon. Below the channel info, there are three tabs: "INÍCIO", "VÍDEOS", and "PLAYLISTS". A "REPRODUIR TODOS" button is also present. The main video thumbnail is titled "Vue.js do jeito ninja" and shows a preview of the course content. Below the thumbnail, two video thumbnails are shown: "Aula 1 - Instalando Vue.js e aprendendo sobre..." and "Aula 2 - O que são diretivas no Vue.js?". Both thumbnails feature the same man, Tiago Matos, and show their respective durations: 7:45 and 4:13. At the bottom of the channel page, there are two small sections: "Tiago Matos 57 mil visualizações • 3 anos atrás" and "Tiago Matos 20 mil visualizações • 3 anos atrás".

Além disso, meu **curso gratuito de Vue.js do jeito ninja** possui mais de 200 mil visualizações no YouTube - desde o seu lançamento, em 2017, vem ajudando vários programadores a conquistarem vagas de emprego no mercado de trabalho.

# **Meu maior objetivo na internet é ajudar você a aumentar o seu repertório e se destacar entre os programadores!**

Por isso, se surgir alguma dúvida, é só perguntar lá no direct do meu instagram: **@tiagomatosweb** ou entrar na nossa comunidade do **Discord**.

Aproveita também para me seguir e ficar por dentro de várias outras dicas sobre programação web, além de acompanhar os bastidores da minha rotina.

Vamosss 

# Peraí! 🤏

**Antes de continuar, quero te fazer um convite.**

Para uma melhor organização de suporte e atenção a você, eu criei um grupo exclusivo no Discord onde poderemos conversar sobre basicamente tudo de tecnologia.

O nosso Discord é um espaço para você tirar dúvidas, compartilhar conhecimentos e conversar sobre programação em geral. E também mais uma forma de ter contato mais próximo e rápido comigo. 🔥

Ah! Lá no Discord rola alguns eventos interessantes como lives, sessão de tira-dúvidas, sorteios e tudo isso ao vivo comigo.

Não vai perder, né?!



Para entrar, basta clicar no link abaixo:



<https://discord.tiagomatos.com>

Vamos nessa?! Conto com sua presença 😊



# Criando uma situação de exemplo

Para que tenhamos maior proveito nesse e-book, eu preciso criar uma situação de exemplo que represente o mundo real da programação para que as 7 dicas masters e, ainda, os 10 bônus imperdíveis sobre técnicas escondidas do Laravel Eloquent façam sentido.

Então, para efeito de exemplificação, vamos criar um e-commerce. Para não ficar muito complexo, focaremos apenas na relação entre **User** e **Order**.

Então, temos os seguintes **models**:

## 01. O model User que possui muita Order



```
<?php  
class Order extends Model  
{  
    public function user()  
    {  
        return $this->belongsTo(User::class);  
    }  
}
```

## 02. O model Order que pertence a um único User



```
<?php  
class User extends Model  
{  
    public function orders()  
    {  
        return $this->hasMany(Order::class);  
    }  
}
```

 **DICA MASTER 1:**

# Use múltiplos arquivos de rotas

Você sabia que nem todas as rotas do seu sistema precisam estar dentro de um só arquivo?

Isso porque, quanto maior for o seu sistema, mais inchado será seu arquivo de rotas, o que pode dificultar bastante o trabalho, sobretudo as manutenções.

Por exemplo, uma vez criei um sistema que tinha mais de 1000 linhas no arquivo de rotas. 😱 Agora, imagine o trabalho que seria encontrar uma rota específica, ou, simplesmente, adicionar uma nova rota no meio disso tudo?

Se você ainda não é tão familiarizado com frameworks, devo dizer que o arquivo de rota é basicamente o ponto de partida do ecossistema (excluindo, obviamente, tudo o que acontece por trás dos bastidores, né?! 😊).

O Laravel, hoje na versão 8, traz por padrão dois arquivos de rotas. O `routes/web.php` para rotas web e `routes/api.php` para rotas da API. E o que isso quer dizer? Isso quer dizer que podemos e devemos criar múltiplos arquivos de rotas.

Então, para uma melhor organização e manutenção, **agrume as rotas relacionadas e as divida em arquivos menores**.

Como? Muito simples! A gente tem duas formas para atingir o mesmo resultado.

## Primeira forma:

- 01.** Abra o arquivo `app/Providers/RouteServiceProvider.php`
- 02.** Localize o bloco semelhante ao que temos abaixo



```
Route::middleware('web')  
    ->namespace($this->namespace)  
    ->group(base_path('routes/web.php'));
```

- 03.** Altere para:



```
Route::middleware('web')  
    ->namespace($this->namespace)  
    ->group(function() {  
        require base_path('routes/users.php');  
        require base_path('routes/orders.php');  
    });
```

Observe que alteramos o *closure* de `group()` para receber uma *function*, em vez de uma *string* como originalmente. Agora, neste caso, a gente “chama” nossos arquivos de rotas usando o *require* do próprio PHP.

Dessa forma, adicionamos 2 arquivos de rotas - um para tratar somente com os *users* e outro para as *orders*. Você pode criar quantos arquivos desejar e dar o nome que melhor se adeque ao seu contexto.

## Segunda forma:

A segunda forma é mais simples. Basta abrir o arquivo de rota original, que pode ser o `routes/web.php` ou `routes/api.php` - o que for mais adequado para o seu cenário - e incluir o código abaixo, passando o caminho do seu arquivo de rota.



```
Route::prefix('users')->group(base_path('routes/users.php'));
Route::prefix('orders')->group(base_path('routes/orders.php'));
```

\*Note que neste caso é necessário declarar o prefixo antes.

Agora você pode declarar as rotas para *users* e *orders* normalmente, dentro de cada arquivo separado.

 **DICA MASTER 2:**

# Use migrations

Os arquivos de *migrations* são uma forma de **controle de versões para a estrutura de banco de dados**. Esses arquivos contém instruções especiais para serem executadas pelo Laravel, através do comando `php artisan migrate`.

Com essa técnica, você pode fazer qualquer tipo de modificação na estrutura do banco de dados através de arquivos, usando a linguagem PHP. Além disso, a partir das migrations, você consegue manipular bancos de dados sem se preocupar com o que já foi feito no passado, uma vez que o Laravel mantém salvo todos os registros de alterações.

Mesmo com a maior parte dos programadores não utilizando a técnica por considerá-la mais burocrática do que a forma convencional, o benefício em curto e longo prazo é incomparável.

Olhando isoladamente é muito mais fácil e rápido você ir direto no banco de dados e fazer as alterações necessárias manualmente, mas no fim das contas, isso pode ser um “tiro no pé”. Por que? Venha comigo...

**Imagine que você tenha que implementar 3 novas funcionalidades independentes (A, B e C) no nosso e-commerce.**

Agora, digamos que em cada uma dessas funcionalidades você precise manipular a estrutura do banco de dados para acomodar a nova funcionalidade. Por exemplo: na funcionalidade A você precisa adicionar uma nova coluna `is_active`, na tabela `users`; já na funcionalidade B, você precisa adicionar uma nova tabela `products`; e, por fim, na funcionalidade C você deve renomear a coluna `total_price` para `amount`, na tabela `orders`.

Só de ler isso, tenho certeza que você já se perdeu e terá que ler novamente para saber o que é o que. Agora imagine depois de um certo tempo trabalhando nessas funcionalidades?

Bom, com apenas as funcionalidades A e B finalizadas, você vai olhar pro banco de dados e se perguntar: *"Êta, quais foram as alterações que fiz no banco de dados para essas duas novas funcionalidades?"*.

A menos que tenha anotado em algum lugar, seu trabalho pode ter ido por água abaixo! Sem falar que terá que replicar as mesmas alterações que fez no seu ambiente de desenvolvimento, também no ambiente de produção.

E se eu adicionar mais programadores trabalhando junto com você? Aí a coisa piora de vez. Haha!

Por outro lado, caso você ou sua equipe utilize os arquivos *migrations*, vocês serão capazes de **visualizar todas as modificações realizadas para cada funcionalidade criada, seja ela A, B ou C**, independente de quem as fez.

E quando tiver que publicar a alteração, basta executar um comando simples que o próprio Laravel executará as instruções contidas nos arquivos *migrations*.

Por exemplo: a instrução abaixo diz ao Laravel para adicionar a coluna `is_active` na tabela `users` sem haver nenhum efeito colateral para o processo de manipulação da estrutura do banco de dados.

```
Schema::table('users', function (Blueprint $table) {  
    $table->boolean('is_active');  
});
```

Assim que você executar o comando `php artisan migrate` no seu servidor de produção, o Laravel executará automaticamente as instruções contidas neste arquivo.

Além disso, se estas alterações não forem realmente as que você planejou, você pode dar um `rollback` para voltar ao estado anterior sem nem mesmo tocar no banco de dados. Para isso, basta usar o comando `php artisan migrate:rollback`.

**Pode ser um pouco mais burocrático, mas percebe como é bem mais vantajoso? 😊**

 **DICA MASTER 3:**

# Use seeding

Seeding é uma habilidade do Laravel de **popular o banco de dados com dados de teste**, possibilitando que o programador monte um ambiente de desenvolvimento já com dados que simulem o funcionamento real do sistema.

Afinal, um banco de dados ao ser criado não possui informações salvas, sendo difícil para qualquer um entender o comportamento do sistema.

**Para ficar mais claro, darei mais um exemplo.** Suponha que você entre para o nosso time de programadores. Em seu primeiro dia, você faz o *clone* do projeto do e-commerce no seu ambiente de desenvolvimento para se familiarizar com o sistema.

Agora imagine que em nosso e-commerce tem uma área administrativa restrita por senha para, literalmente, administrar todo o sistema. Coisas do tipo: listar todos os usuários, ver o que cada usuário comprou, listar pedidos, etc.

Só que, o nosso e-commerce precisa de usuário e senha para acesso e o seu banco de dados local está vazio, pois o nosso time não utiliza a técnica de seeding. Além disso, sua tabela `users` e `orders` também estão vazias.

Para solucionar isso, você precisa ir no banco de dados e, manualmente, adicionar um usuário administrador para possibilitar o acesso ao sistema, adicionar alguns usuários clientes - aqueles que

fazem a compra no e-commerce, e, finalmente, adicionar alguns pedidos na tabela `orders` para poder ter dados visíveis no painel administrativo.

E olhe que estamos falando apenas de praticamente dois `models`, `User` e `Order`, mas, sabemos que na vida real temos muito mais que isso, a exemplo de `Products`, `Categories`, `Roles`, etc.

E aí? **Você vai criar tudo manualmente?** Um por um? 😱

*Vai chegar no final do mês e você não terá conseguido ainda rodar o sistema completo no seu ambiente de desenvolvimento. Haha!*

É aí que entra a técnica de seeding!

Uma vez que, ao popular o banco de dados com aqueles dados de teste, o ambiente de desenvolvimento irá se comportar como se fosse o ambiente de produção. Logo, a vantagem da técnica de seeding é que o desenvolvedor conseguirá rodar o sistema facilmente, uma vez que ele tem todo o banco de dados preenchido com dados de teste que representam o mundo real do sistema.

Para criar arquivos de seeds, basta utilizar o comando `php artisan make:seed` seguido do nome do arquivo desejado. Por exemplo, `php artisan make:seed UserSeeder` para gerar o arquivo `database/seeders/UserSeeder.php` e `php artisan make:seed OrderSeeder`, para gerar o arquivo `database/seeders/OrderSeeder.php`.

E ainda com o Laravel, você pode utilizar o comando da `migrations` que vimos na dica anterior, passando uma opção `--seed` que chamamos de `flag`, que irá informar ao Laravel para alimentar o banco de dados através da técnica de seeding. O comando completo seria `php artisan migrate --seed`.

 DICA MASTER 4:

# Não use validação no controller

Lembra das rotas que eu falei lá na primeira dica? Com certeza você também deve saber que posso enviar dados para essas rotas, via o método `POST`, para ser lido pelo *controller*. Porém, antes de serem salvos no banco de dados, esses dados precisam ser *validados*.

Programadores tendem, naturalmente, a declarar as regras de validação dentro do próprio *controller*, principalmente, os programadores iniciantes.



```
class OrderController extends Controller
{
    public function store(Request $request)
    {
        $request->validate([
            'title'      => 'required|string',
            'products'   => 'required|string',
            'notes'       => 'string',
        ]);
    }
}
```

**Isso não está errado, porém é ineficaz.**

**Ao utilizar essa estratégia, você não é capaz de reutilizar essa validação em outros **controllers**, caso seja necessário.**

Neste caso, você tem de copiar e colocar o bloco de validação em outros locais, manualmente. E é aí que mora o problema!

E se eu quiser alterar a validação do campo **title**, por exemplo? Eu tenho que ir em todos os locais que utilizam essa mesma validação para alterar o que desejo, de novo, manualmente.

**Ou seja, não é uma forma inteligente e reutilizável.** E nós, programadores, gostamos de reutilização, não é mesmo?!

Por isso, em vez da estratégia de validação dentro do **controller**, eu crio a validação em cima da requisição concentrada em um só arquivo, que chamamos de **Form Request Validation**.

Este arquivo nada mais é do que uma simples classe PHP que estende a classe **FormRequest** do Laravel e que possui um método chamado **rules()**, onde constam as regras de validação. Então, vamos criar um arquivo chamado **StoreOrderRequest.php** para implementar essa mesma validação.

Neste arquivo, eu consigo incluir as mesmas regras que eu utilizo lá dentro do **controller**, mas de uma forma isolada e flexível. Vejamos:

```
class StoreOrderRequest extends FormRequest
{
    public function rules()
    {
        return [
            'title'      => 'required|string',
            'products'   => 'required|string',
            'notes'       => 'string',
        ];
    }
}
```

Agora, basta eu injetar este arquivo no método do controller que o Laravel vai fazer toda a mágica por detrás dos bastidores e aplicar a validação que eu defini.

Fica assim:

```
class OrderController extends Controller
{
    public function store(StoreOrderRequest $request)
    {
        //
    }
}
```

E agora eu posso facilmente reutilizar essa mesma validação em qualquer outro lugar, sem precisar ficar repetindo código.

Além disso, caso eu precise alterar alguma regra nessa validação que foi utilizada em várias partes do sistema, não preciso fazer a modificação de um a um. Basta alterar um arquivo, o `StoreOrderRequest.php`, que todos os outros lugares respeitarão as novas regras de validação.

 DICA MASTER 5:

# Manipulação de erros customizada

O Laravel é tão inteligente que por padrão ele já faz o tratamento de erros pra você. Erros são chamados mais comumente de **exceptions**, certo?

Todas as **exceptions** são jogadas para o arquivo `app/Exceptions/Handler.php` e de lá, você pode manipular cada **exception** para ter um comportamento diferenciado. O arquivo `Handler.php` possui um método `register()` que é onde você faz suas customizações.

Vamos supor que queremos tratar um erro quando uma **Order** for inválida.



```
public function register()
{
    $this->reportable(function (InvalidOrderException $e) {
        //
    });
}
```

Depois de várias inserções de diferentes tipos de ***exceptions***, esse arquivo fica muito inchado e praticamente ilegível, o que dificulta bastante a manutenção.

Imagine este arquivo com tratamento de cerca de 100 tipos de ***exceptions*** diferentes? 😱

Assim, a solução que eu uso é: **criar arquivos separados para cada tipo de exception**.

Esse arquivo de ***exception*** nada mais é do que uma classe PHP que estende a própria classe **Exception** do PHP e que possui um método especial chamado de **render()**, onde inserimos a lógica para este erro em específico.

```
use Exception;

class InvalidOrderException extends Exception
{

    public function render()
    {
        //
    }
}
```

# Pronto! É somente isso.

É sério. Haha! 😊

A partir de agora, sempre que usar esta **exception** através do comando `throw new InvalidOrderException()`, o Laravel magicamente reconhece essa classe e lê o método `render()` e então, executa o que você tiver definido lá.

Além disso, assim como na validação, essa tática de separar os erros em diferentes arquivos possibilita que você reutilize em várias outras partes do sistema, acelerando o processo de programação.

 **DICA MASTER 6:**

# Pare de usar o blade template

Calma! Antes que você queira me enforcar, eu tenho um grande motivo para te dizer para não usar o **Blade Template**.

Você sabe que o **Blade** é um mecanismo do Laravel utilizado para criar o template das páginas. Logo, não é errado usar esse mecanismo, uma vez que é ofertada pelo próprio Laravel.

*Ah, sim! Vamos deixar as coisas claras aqui que eu confesso que o Blade é muito poderoso e flexível.*

Mas então, por que eu estou te aconselhando a **parar** de usar o Blade?

Se você já é mais avançado no mundo Laravel, sabe muito bem que esse framework oferece praticamente todos os recursos para criar uma aplicação completa. Com o Laravel, você pode criar funcionalidades para backend e frontend em um único projeto.

Usando Blade para trabalhar com a camada **view** da sua aplicação, faz com que o seu projeto vire um sistema monolítico, ou seja, frontend e backend em um único projeto. Nada de errado com isso, mas é aí que mora um certo limite de flexibilidade!

 **Então a dica aqui é:** desacople o frontend do backend criando projetos diferentes para ambos. Trocando em miúdos, isso é o que chamamos de **arquitetura API** - frontend separado do backend e se comunicando através de requisições HTTP. O benefício é incomparável!

Não vou entrar em muitos detalhes aqui sobre sistemas monolíticos e arquitetura API, mas, a principal razão de se usá-las é que essa arquitetura pode prover informações para múltiplos frontends, concentrando todas as regras de negócio em um único ponto.

A API pode alimentar o nosso e-commerce, um futuro app e ainda, pode se integrar facilmente com outros sistemas externos do tipo meio de pagamento como MercadoPago e Paypal.

Com isso em mente, eu recomendo altamente que, em vez do Blade, você utilize um outro framework dedicado para o seu frontend, como por exemplo os frameworks javascript: **Vue.js, Reactjs ou Angular**.

 DICA MASTER 7:

# Use sempre eager loading

Ao acessar relacionamentos do Eloquent como propriedades, os **models** relacionados são lidos em tempo de execução. Isso significa que os dados de relacionamento não são literalmente carregados até que você acesse a propriedade pela primeira vez.

Isso é muito bom, pois se você tiver muitos relacionamentos, o Laravel não carregará até que você os solicite. Porém, em alguns casos, essa estratégia pode ir por água abaixo.

Para ilustrar, vamos exibir o nome do usuário de todas as orders registradas no nosso sistema. Lembre-se que cada **Order** pertence a um único **User**.

A princípio somos tendenciosos a fazer isso aqui:

```
$orders = Order::all();

foreach ($orders as $order) {
    echo $order->user->name;
}
```

## E qual o problema?

O problema é que o Laravel não carrega a relação `Order->user()` inicialmente. Os dados dessa relação são carregados apenas quando você os solicita, em tempo de execução. Neste caso, na linha `$order->user->name`.

E de novo, qual o problema disso? O problema é que a cada execução uma nova consulta é feita ao banco de dados e isso pode se tornar muito custoso para o seu sistema.

Imagine que temos cerca de 1000 orders. Então serão 1001 consultas, em 4 linhas de códigos. Serão 1000 consultas para pegar o nome do usuário de cada `Order` e uma consulta inicial para pegar todas as `orders`.

Percebe agora o tamanho do problema?

Para resolver isso, basta utilizar a técnica Eager Loading adicionando `with('user')` na nossa consulta, dessa forma:



```
$orders = Order::with('user')->get();
```

`user` aqui é justamente o nome da relação que a gente definiu lá no *model Order*.

Assim, o Laravel irá incluir em uma única consulta o pedido de carregamento da relação `Order->user()`, resultando em apenas um consulta, e não 1001.



## BÔNUS

# 10 dicas rápidas de Laravel Eloquent que você provavelmente não sabe!



### DICA BÔNUS 1

O método `get()` e `all()` podem aceitar um `array` para retornar colunas específicas, ou uma `string` para retornar apenas uma coluna em específico.



```
User::where('id', 1)->get('name');  
User::where('id', 1)->get(['name', 'is_active']);  
  
User::all('name');  
User::all(['name', 'is_active']);
```



### DICA BÔNUS 2

Combinação da condição `where`

Em vez disso:



```
Order::where('user_id', 1)->where('status', 'pending')->get();
```

Você pode fazer isso:



```
Order::whereUserIdAndStatus(1, 'pending')->get();
```

### 🚀 DICA BÔNUS 3

Outra forma de usar *dd*

Em vez disso:



```
$order = Order::find(1);  
dd($order);
```

Você pode fazer isso:



```
$order = Order::find(1)->dd();
```

### 🚀 DICA BÔNUS 4

Incrementar e decrementar uma coluna.

Em vez disso:



```
$order = Order::find(1);  
$order->views++;  
$order->save();
```

Você pode fazer isso:

```
● ● ●  
$order = Order::find(1);  
$order->increment('views');  
$order->decrement('views');
```

E ainda isso para incrementar/decrementar em certa quantidade:

```
● ● ●  
$user = User::find(1);  
$user->increment('points', 10);
```

### 🚀 DICA BÔNUS 5

O método find aceita um array de id's

```
● ● ●  
$order = Order::find([1, 2, 3]);
```

## DICA BÔNUS 6

Eloquent::when()

É normal e nada de errado criar algumas condições para as *queries* dessa forma:

```
$orders = Order::with('user');

if (!empty(request(key: 'keyword'))) {
    $orders->where('description', 'LIKE', '%' . request(key: 'keyword') . '%');
}

if (!empty(request(key: 'user_id'))) {
    $orders->where('user_id', request(key: 'user_id'));
}
```

Mas que tal você escrever a mesma query de uma forma mais elegante?

```
$orders = Order::with('user');

$orders->when(request()->has('keyword'), function ($q) {
    $q->where('name', 'LIKE', '%' . request(key: 'keyword') . '%');
});

$orders->when(request()->has('user_id'), function ($q) {
    $q->where('user_id', request(key: 'user_id'));
});
```

Perceba que o callback só é executado quando o primeiro parâmetro for **true**.

## DICA BÔNUS 7

Duplicando um model

```
$order = Order::find(1);  
$newOrder = $order->replicate();  
$newOrder->save();
```

## DICA BÔNUS 8

Trabalhando com grande volume de registros

Não é exatamente relacionado ao Eloquent, mas muito útil para trabalhar com grande volume de registros. O método `chunk()` retorna pequenos pedaços de resultados.

```
Order::chunk(100, function ($orders) {  
    foreach ($orders as $order) {  
        //  
    }  
});
```



## DICA BÔNUS 9

`findOrFail / firstOrCreate`

Em vez disso:

```
$user = User::find(1);  
if (!$user) { abort ( code:404); }
```

Você pode fazer isso:

```
$user = User::findOrFail(1);
```

Em vez disso:

```
$user = User::where('email', $email)->first();  
if (!$user) {  
    User::create([  
        'email' => $email  
    ]);  
}
```

Você pode fazer isso:



```
$user = User::firstOrCreate(['email' => $email]);
```



## DICA BÔNUS 10

BelongsTo default methods

Como já abordamos, nós podemos exibir o nome do usuário de uma **Order** dessa forma:



```
{{ $order->user->name }}
```

Mas se por acaso o usuário não existir por alguma razão, teremos um erro aí. Provavelmente o PHP irá exibir algo do tipo '*property of non-object*'.

A gente pode fazer isso sempre que for buscar o nome do usuário:



```
{{ $order->user->name ?? '' }}
```

Mas, você pode tornar isso como padrão, alterando a relação user lá no model **Order**, adicionando **withDefault()**:

```
public function user()  
{  
    return $this->belongsTo(User::class)->withDefault();  
}
```

Agora, por padrão, user será sempre vazio quando o mesmo não existir.

# Conclusão

**Viu só, quanta dica massa você estava perdendo?**



Comece a colocá-las em prática, e logo você verá a diferença nos seus projetos Laravel, tanto na criação quanto na manutenção.

Quando começar a usar essas dicas, compartilhe comigo no meu instagram, **@tiagomatosweb!**

Quero saber se essas táticas estão salvando a sua vida da mesma forma que salvam a minha.

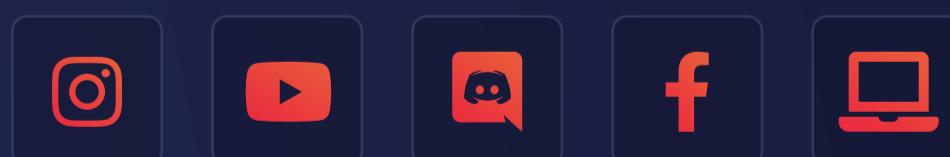
E claro, aproveite para seguir o meu perfil e ficar por dentro de todos os materiais e cursos disponíveis.

Um forte abraço jovem e vamos decolar A row of three small, colorful rocket ship icons pointing upwards.



Tiago Matos

## Meus links [🔗](#)



## Referencias

Business vector created by freepik - [www.freepik.com](http://www.freepik.com)