POO
Programação para a WEB - servidor (server-side)

Sérgio da Silva Nogueira

PHP

ELOQUENT
ORM (Object Relational Mapping)

Eloquent é o nome do ORM (Object Relational Mapping) nativo no Framework PHP Laravel, que facilita toda a manipulação da base de dados. Não importa se é uma simples inserção de dados ou uma procura extremamente complexa com relações entre tabelas, o Eloquent dispõe de métodos de fácil compreensão e conta com uma documentação completa de como funciona dentro do Laravel.

PHP

Models

No Laravel os models são a representação, do ponto de vista de objetos, das tabelas da base de dados. Representação essa, pensando em uma entidade que represente todos os dados da tabela em questão.

Por exemplo, por convenção do framework, se eu tenho uma tabela chamada posts na base, a representação em model desta tabela será uma classe chamada de Post. Se eu tenho uma tabela users sua representação via model será uma classe chamada de User.

Quando nós temos entidades/models no singular o Laravel automaticamente tentará, por convenção, resolver sua tabela no plural por ter o pensamento, na base, de uma coleção de dados.

Por exemplo, como fazer um select a todos os posts via Model?

Para isto temos o método **all** que faz este trabalho por nós

PHP

Models

Por exemplo, como fazer um select a todos os posts via Model?

Para isto temos o método **all** que faz este trabalho por nós

```
Post::all()
```

O resultado do método acima, se fossemos pensar em uma query na base seria uma sql como esta:

select * from posts

Onde o Laravel pegará automáticamente o nome do seu model e tentará resolver ele no plural na

execução da query, por exemplo model Post tabela posts.

PHP

Model Post

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    //
}
```

O model acima, somente com sua definição bem simples inclusive, já permite realizar diversas operações em cima da tabela posts associada ao model Post. Se, por ventura, for necessário utilizar um nome de tabela diferente e não quisermos que o Laravel resolva o nome dela, você pode reescrever o atributo dentro do seu model.

PHP

Model Post

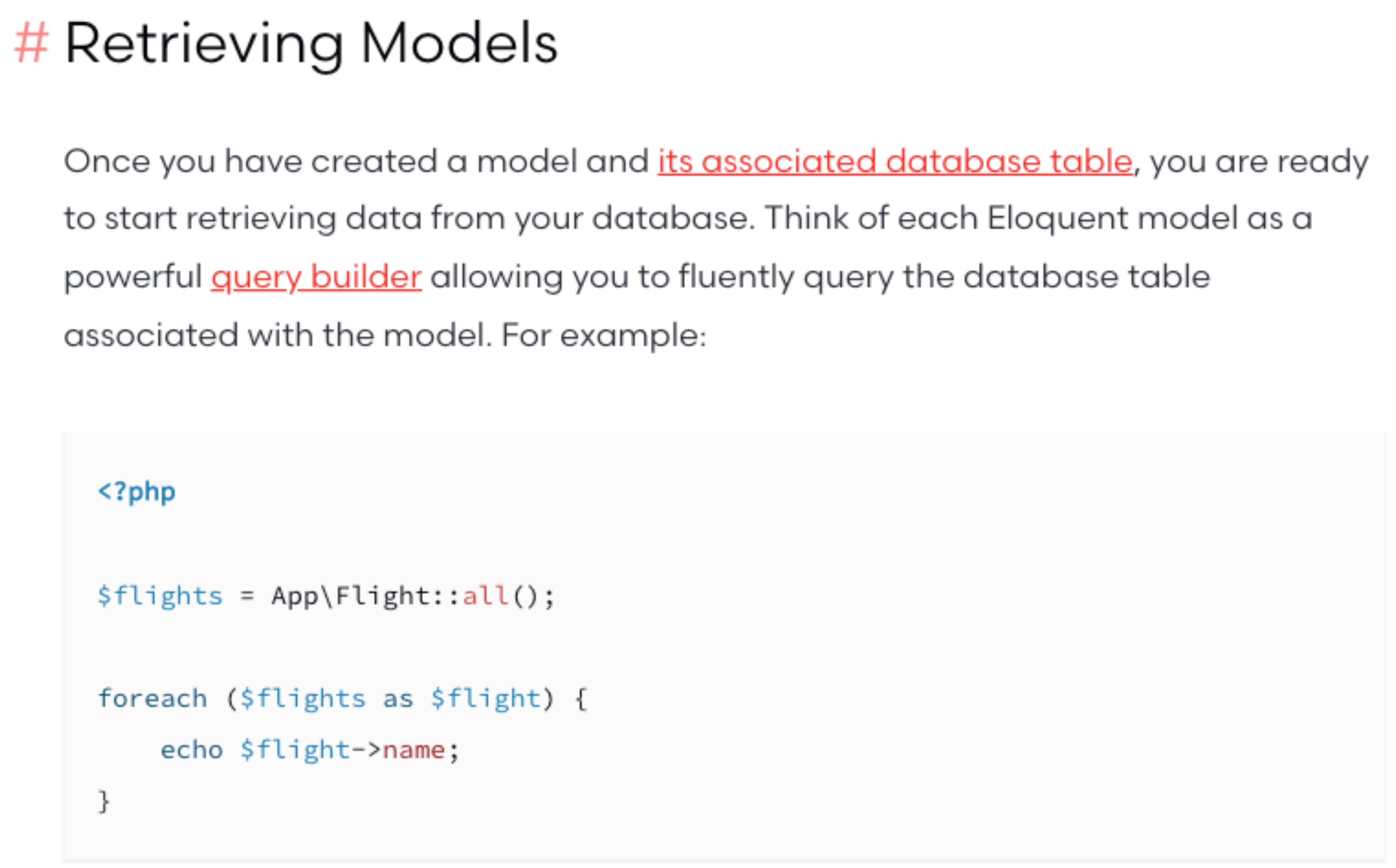
```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $table = 'table_name';
}
```

PHP

Eloquent

Documentação:

https://laravel.com/docs/7.x/eloquent



Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example:

```php
<?php

$flights = App\Flight::all();

foreach ($flights as $flight) {
    echo $flight->name;
}
```

PHP

Eloquent

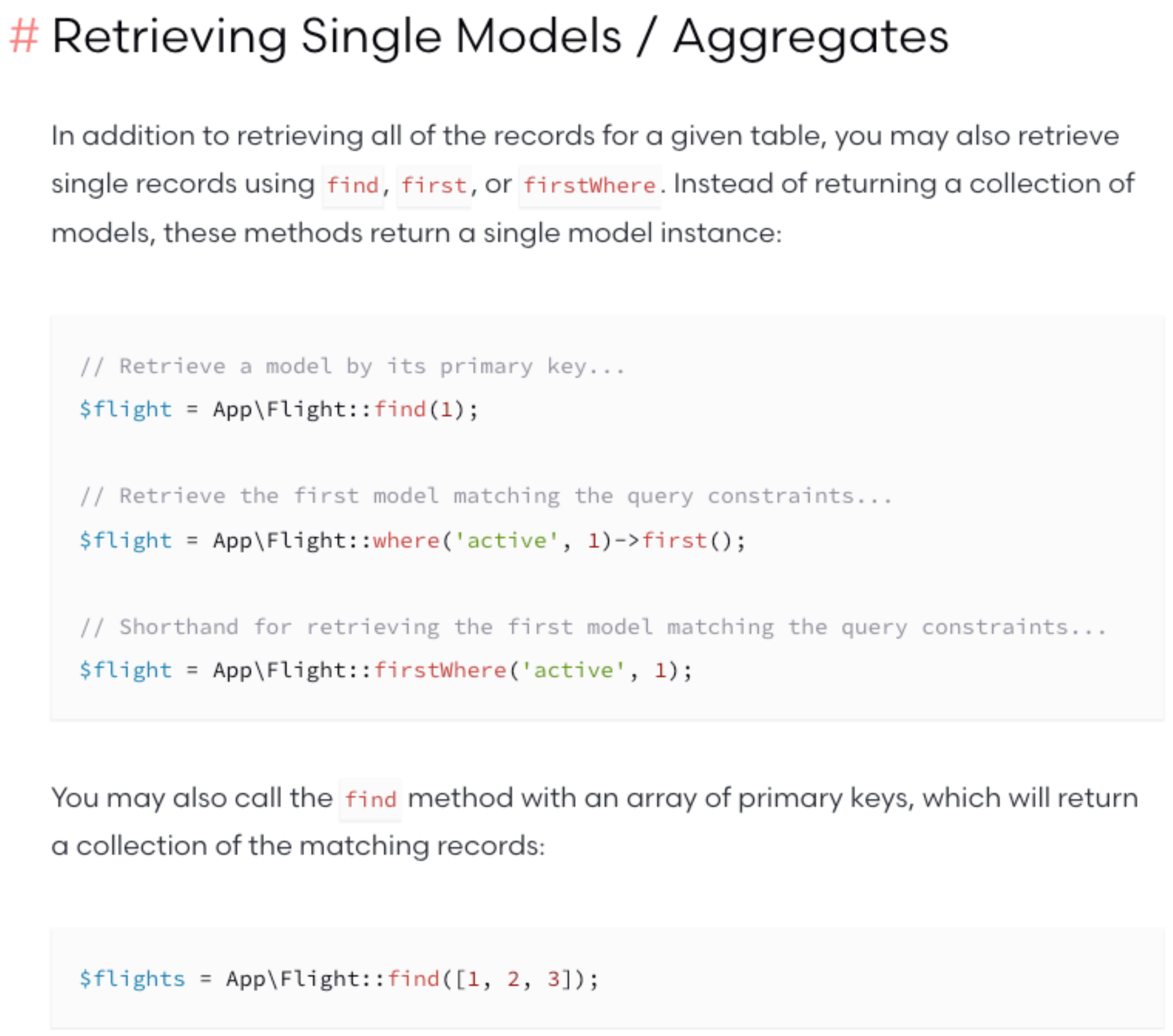
Adding Additional Constraints

The Eloquent `all` method will return all of the results in the model's table. Since each Eloquent model serves as a query builder, you may also add constraints to queries, and then use the `get` method to retrieve the results:

```
$flights = App\Flight::where('active', 1)
               ->orderBy('name', 'desc')
               ->take(10)
               ->get();
```

PHP

Eloquent



Retrieving Single Models / Aggregates

In addition to retrieving all of the records for a given table, you may also retrieve single records using `find`, `first`, or `firstWhere`. Instead of returning a collection of models, these methods return a single model instance:

```
// Retrieve a model by its primary key...
$flight = App\Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = App\Flight::where('active', 1)->first();

// Shorthand for retrieving the first model matching the query constraints...
$flight = App\Flight::firstWhere('active', 1);
```

You may also call the `find` method with an array of primary keys, which will return a collection of the matching records:

```
$flights = App\Flight::find([1, 2, 3]);
```

PHP

Eloquent

Sometimes you may wish to retrieve the first result of a query or perform some other action if no results are found. The `firstOr` method will return the first result that is found or, if no results are found, execute the given callback. The result of the callback will be considered the result of the `firstOr` method:

```
$model = App\Flight::where('legs', '>', 100)->firstOr(function () {
        // ...
});
```

The `firstOr` method also accepts an array of columns to retrieve:

```
$model = App\Flight::where('legs', '>', 100)
            ->firstOr(['id', 'legs'], function () {
                // ...
            });
```

PHP

Eloquent

Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query; however, if no result is found, a `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

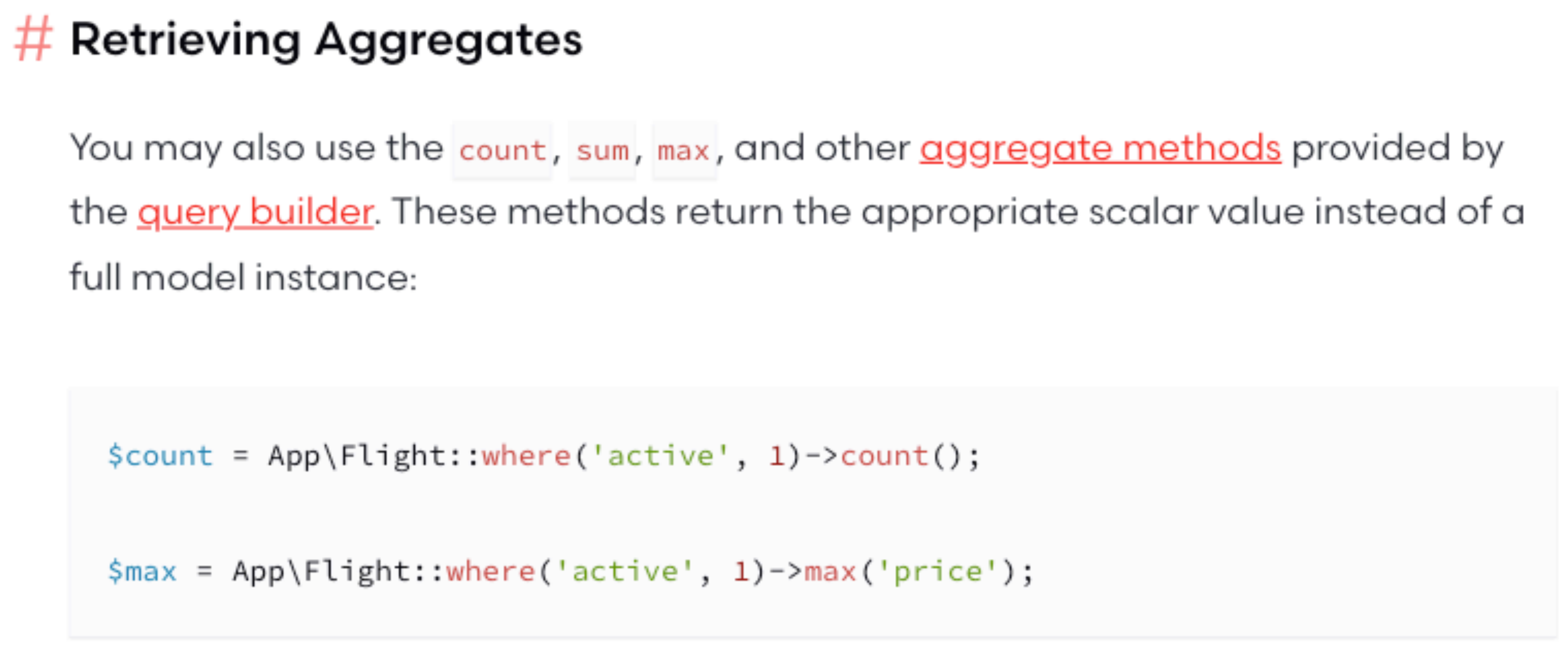
```php
$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

If the exception is not caught, a `404` HTTP response is automatically sent back to the user. It is not necessary to write explicit checks to return `404` responses when using these methods:

```php
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

PHP

Eloquent



Retrieving Aggregates

You may also use the `count`, `sum`, `max`, and other aggregate methods provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

```
$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');
```

PHP

ELOQUENT CONVENÇÕES

Naming Controllers

Controllers should be in singular case, no spacing between words, and end with "Controller".

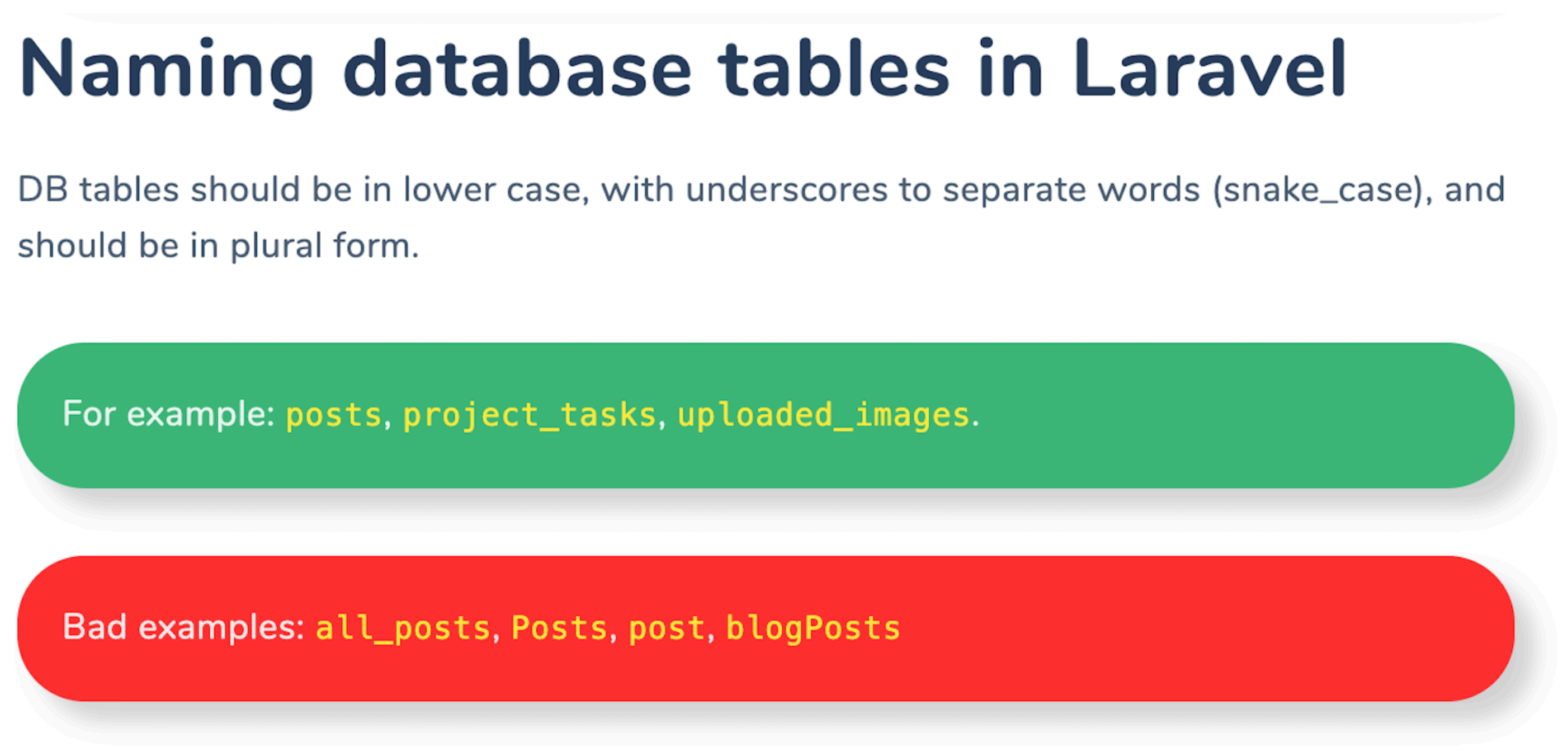
Also, each word should be capitalised (i.e. BlogController, not blogcontroller).

For example: `BlogController`, `AuthController`, `UserController`.

Bad examples: `UsersController` (because it is in plural), `Users` (because it is missing the Controller suffix).

PHP

ELOQUENT CONVENÇÕES



Naming database tables in Laravel

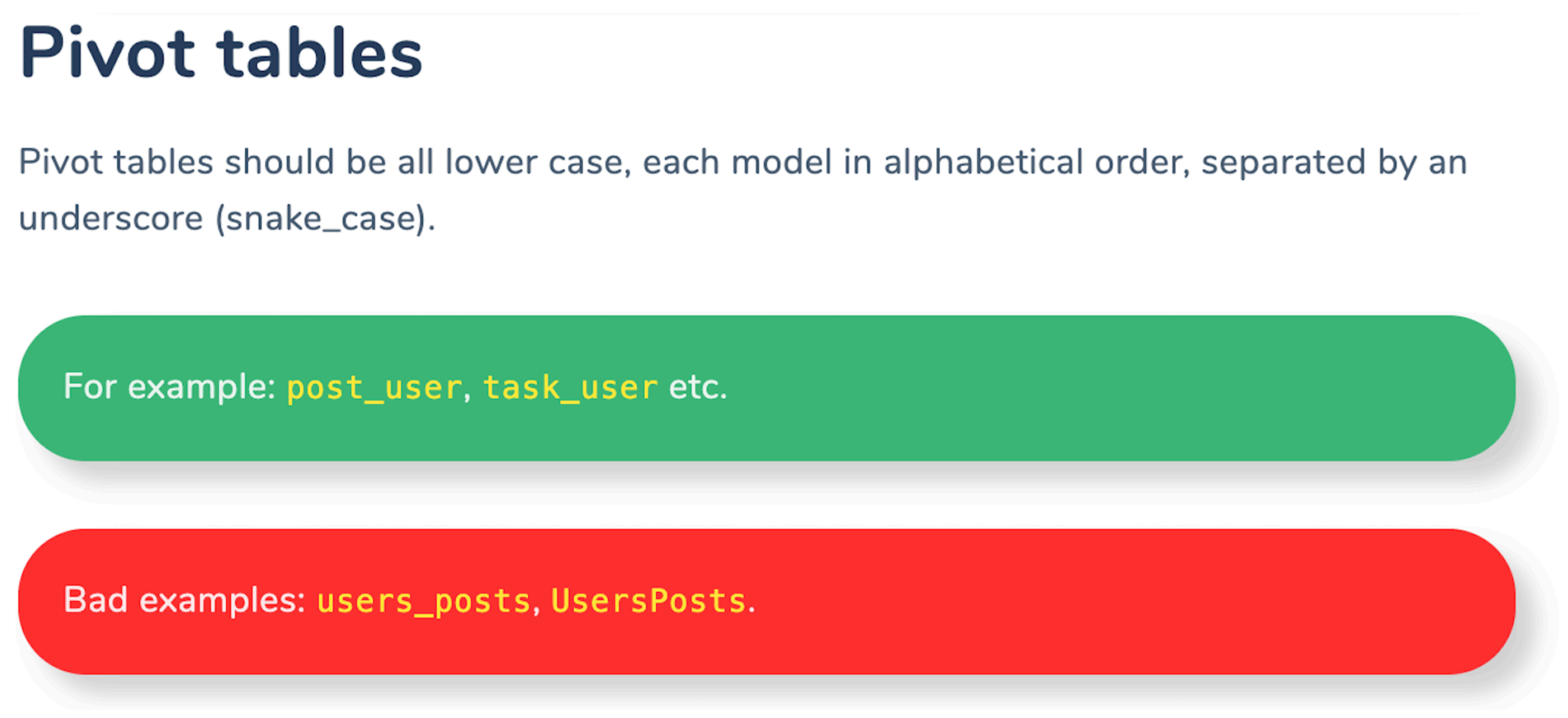
DB tables should be in lower case, with underscores to separate words (snake_case), and should be in plural form.

For example: `posts`, `project_tasks`, `uploaded_images`.

Bad examples: `all_posts`, `Posts`, `post`, `blogPosts`

PHP

ELOQUENT CONVENÇÕES



Pivot tables

Pivot tables should be all lower case, each model in alphabetical order, separated by an underscore (snake_case).

For example: `post_user`, `task_user` etc.

Bad examples: `users_posts`, `UsersPosts`.

PHP

ELOQUENT CONVENÇÕES

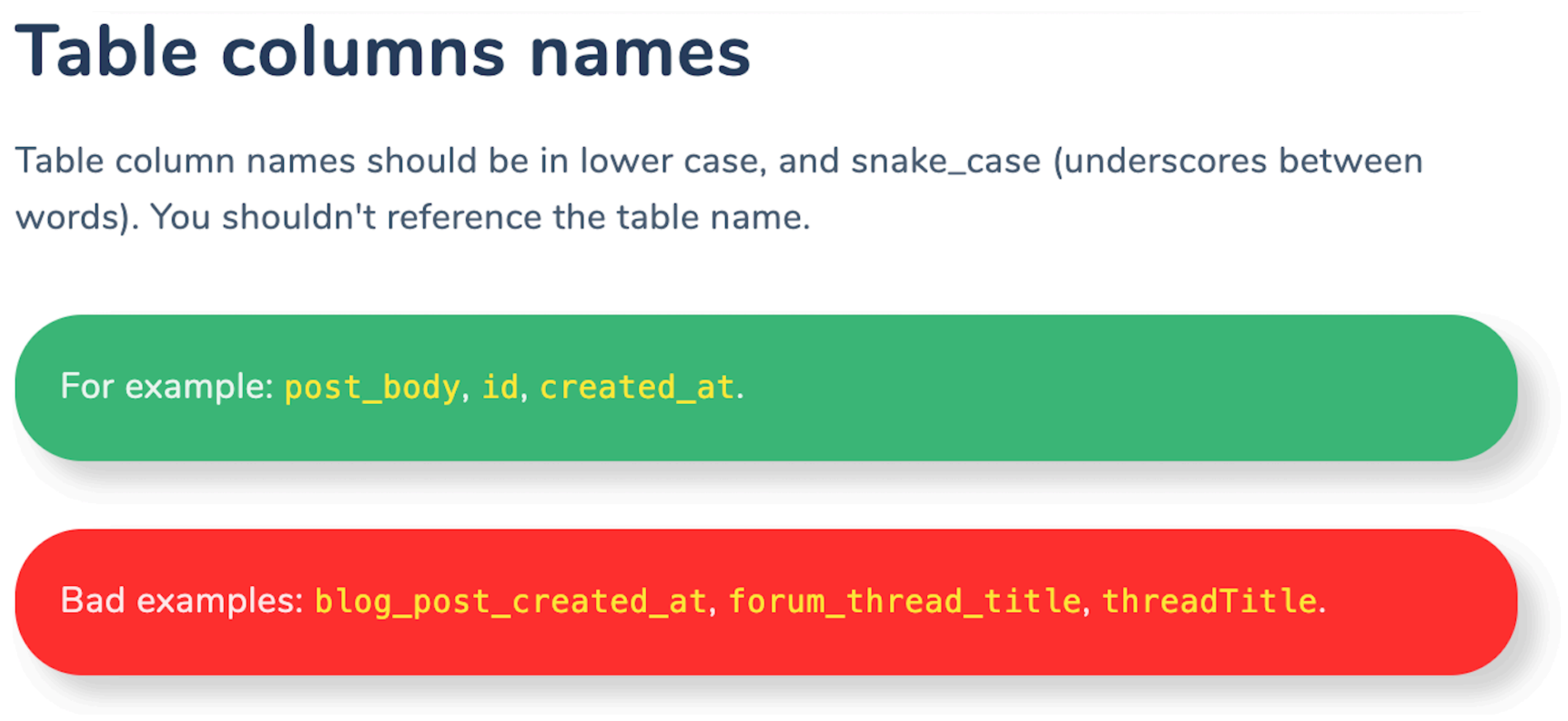


Table columns names

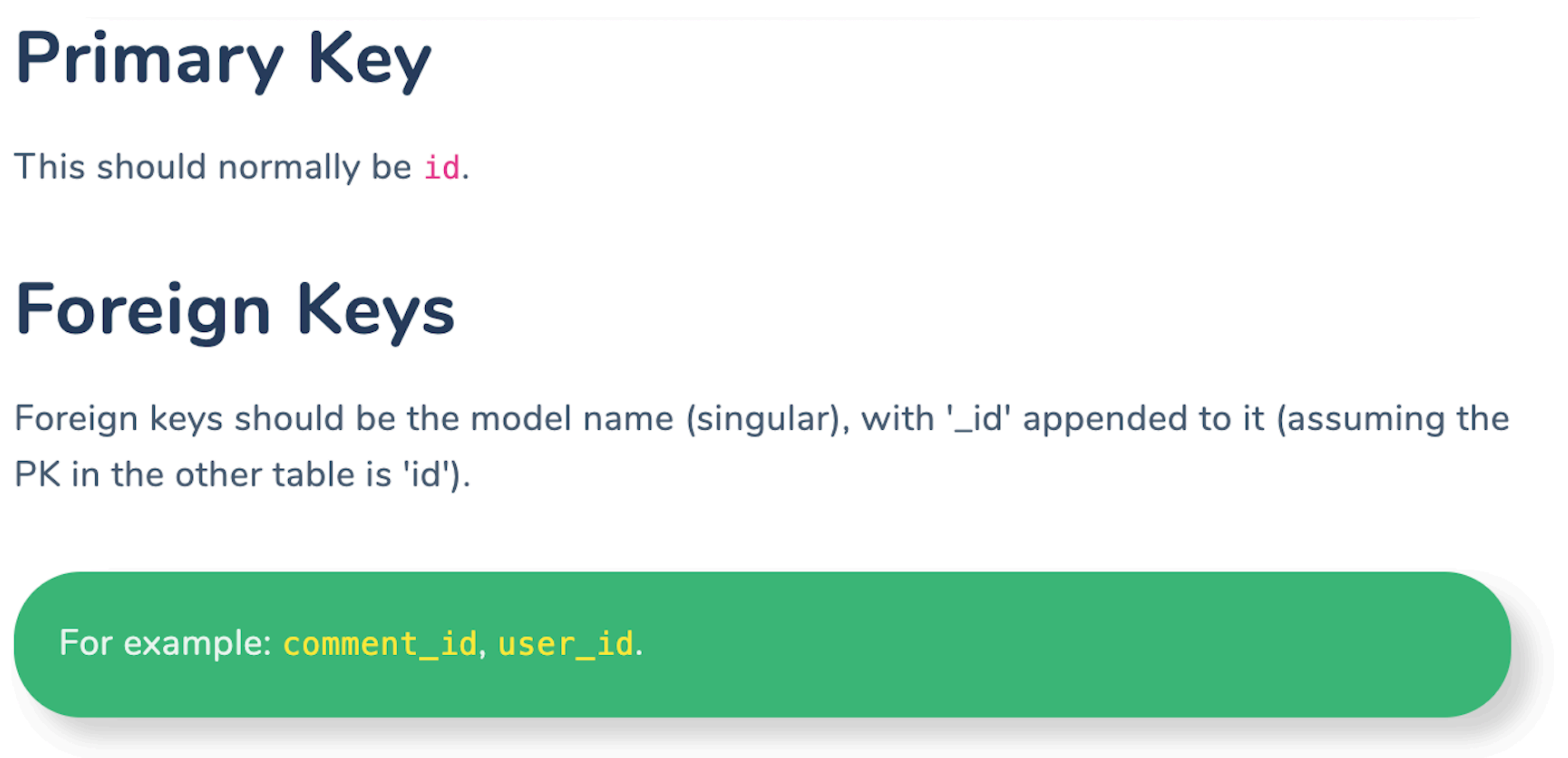
Table column names should be in lower case, and snake_case (underscores between words). You shouldn't reference the table name.

For example: `post_body`, `id`, `created_at`.

Bad examples: `blog_post_created_at`, `forum_thread_title`, `threadTitle`.

PHP

ELOQUENT CONVENÇÕES



Primary Key

This should normally be `id`.

Foreign Keys

Foreign keys should be the model name (singular), with '_id' appended to it (assuming the PK in the other table is 'id').

For example: `comment_id`, `user_id`.

PHP

ELOQUENT CONVENÇÕES

Variables

Normal variables should typically be in camelCase, with the first character lower case.

For example: `$users = ...`, `$bannedUsers = ....`

Bad examples: `$all_banned_users = ...`, `$Users=....`

If the variable contains an array or collection of multiple items then the variable name should be in plural. Otherwise, it should be in singular form.

For example: `$users = User::all();` (as this will be a collection of multiple User objects), but `$user = User::first()` (as this is just one object)

PHP

ELOQUENT CONVENÇÕES

Verb	URI	Typical Method Name	Route Name
GET	`/photos`	`index()`	photos.index
GET	`/photos/create`	`create()`	photos.create
POST	`/photos`	`store()`	photos.store
GET	`/photos/{photo}`	`show()`	photos.show
GET	`/photos/{photo}/edit`	`edit()`	photos.edit
PUT/PATCH	`/photos/{photo}`	`update()`	photos.update
DELETE	`/photos/{photo}`	`destroy()`	photos.destroy

PHP

ELOQUENT Relações

As tabelas da base de dados costumam estar relacionadas entre si. Por exemplo, um Post de um blog pode ter muitos comentários ou um pedido pode estar relacionado com o utilizador que o fez. O Eloquent facilita a gestão e o trabalho com estas relações e oferece suporte aos vários tipos.

PHP

ELOQUENT Relações

One To One

One To Many

One To Many (Inverse)

Many To Many

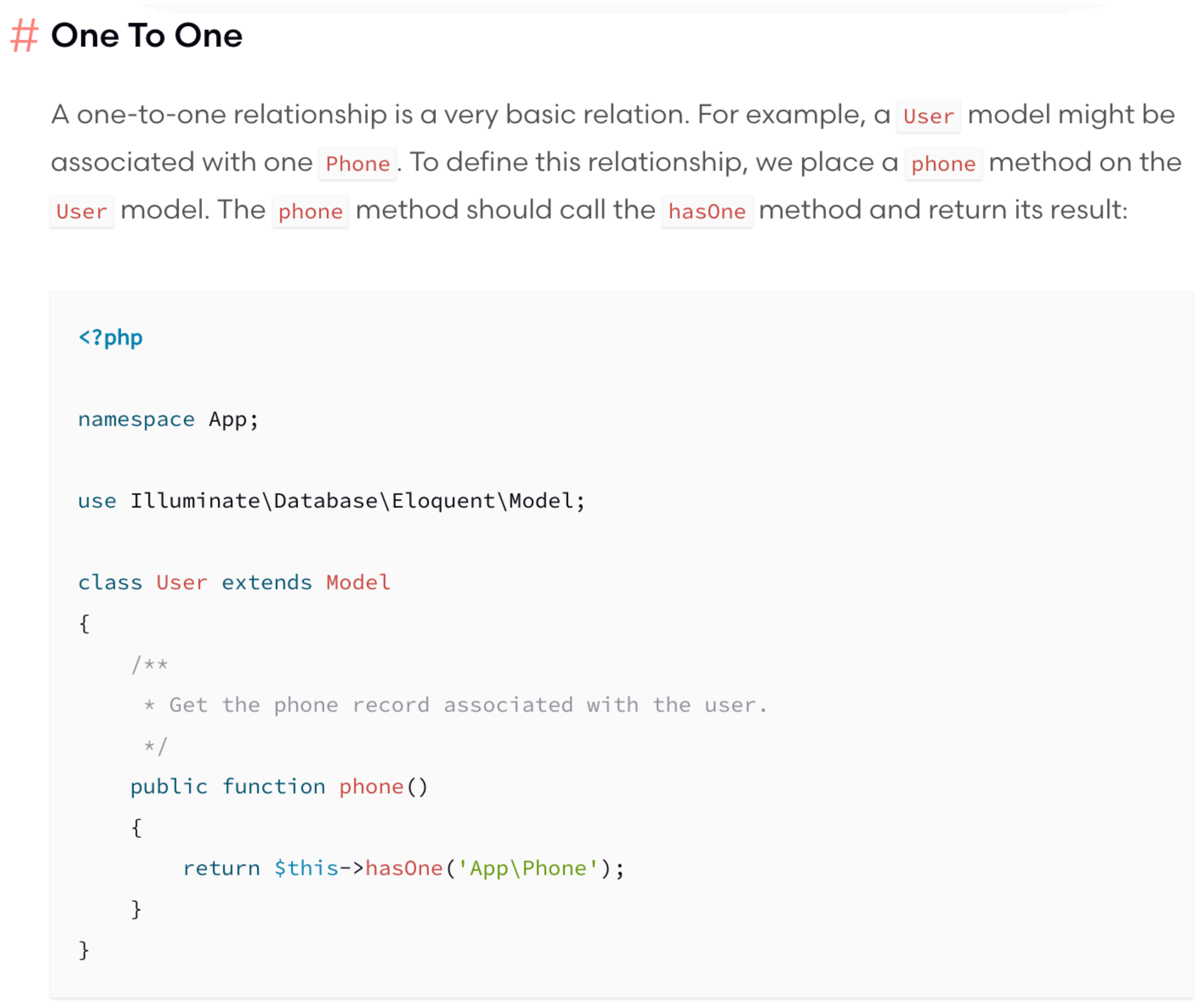
Defining Custom Intermediate Table Models

Has One Through

Has Many Through

PHP

ELOQUENT Relações



One To One

A one-to-one relationship is a very basic relation. For example, a `User` model might be associated with one `Phone`. To define this relationship, we place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the phone record associated with the user.
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

PHP

ELOQUENT Relações

The first argument passed to the `hasOne` method is the name of the related model. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship methods as if they were properties defined on the model:

```php
$phone = User::find(1)->phone;
```

PHP

ELOQUENT Relações

Eloquent determines the foreign key of the relationship based on the model name. In this case, the `Phone` model is automatically assumed to have a `user_id` foreign key. If you wish to override this convention, you may pass a second argument to the `hasOne` method:

```
return $this->hasOne('App\Phone', 'foreign_key');
```

PHP

ELOQUENT Relações

Defining The Inverse Of The Relationship

So, we can access the `Phone` model from our `User`. Now, let's define a relationship on the `Phone` model that will let us access the `User` that owns the phone. We can define the inverse of a `hasOne` relationship using the `belongsTo` method:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

PHP

ELOQUENT Relações

One To Many

A one-to-many relationship is used to define relationships where a single model owns any amount of other models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by placing a function on your Eloquent model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

PHP

ELOQUENT Relações

One To Many (Inverse)

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a `hasMany` relationship, define a relationship function on the child model which calls the `belongsTo` method:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

PHP

ELOQUENT Relações

Many To Many

Many-to-many relations are slightly more complicated than `hasOne` and `hasMany` relationships. An example of such a relationship is a user with many roles, where the roles are also shared by other users. For example, many users may have the role of "Admin".

Table Structure

To define this relationship, three database tables are needed: `users`, `roles`, and `role_user`. The `role_user` table is derived from the alphabetical order of the related model names, and contains the `user_id` and `role_id` columns:

```
users
    id - integer
    name - string

roles
    id - integer
    name - string

role_user
    user_id - integer
    role_id - integer
```

PHP

ELOQUENT Relações

Model Structure

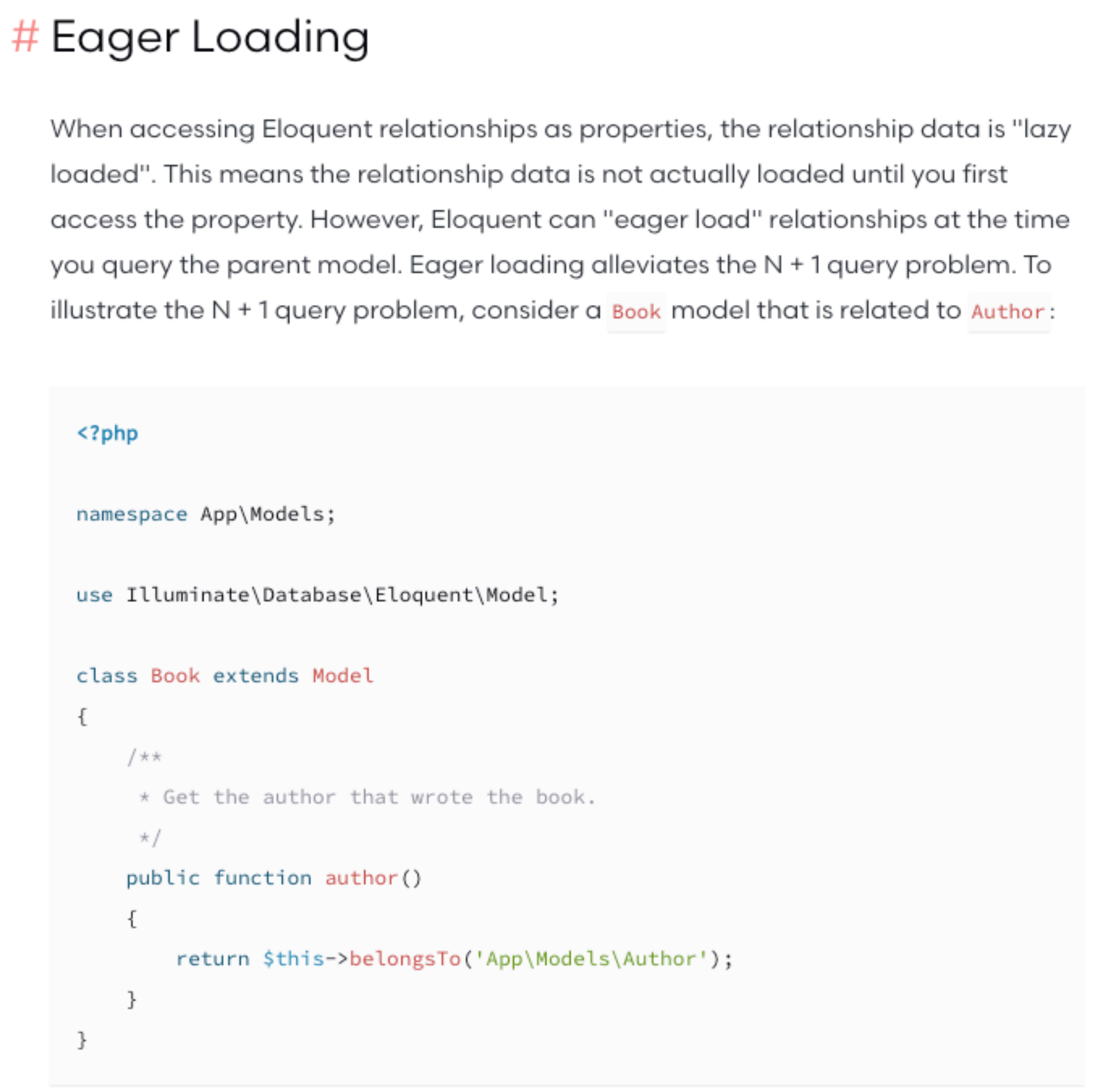
Many-to-many relationships are defined by writing a method that returns the result of the `belongsToMany` method. For example, let's define the `roles` method on our `User` model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

Once the relationship is defined, you may access the user's roles using the `roles` dynamic property:

PHP

ELOQUENT Eager Loading



Eager Loading

When accessing Eloquent relationships as properties, the relationship data is "lazy loaded". This means the relationship data is not actually loaded until you first access the property. However, Eloquent can "eager load" relationships at the time you query the parent model. Eager loading alleviates the N + 1 query problem. To illustrate the N + 1 query problem, consider a `Book` model that is related to `Author`:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo('App\Models\Author');
    }
}
```

PHP

ELOQUENT Eager Loading

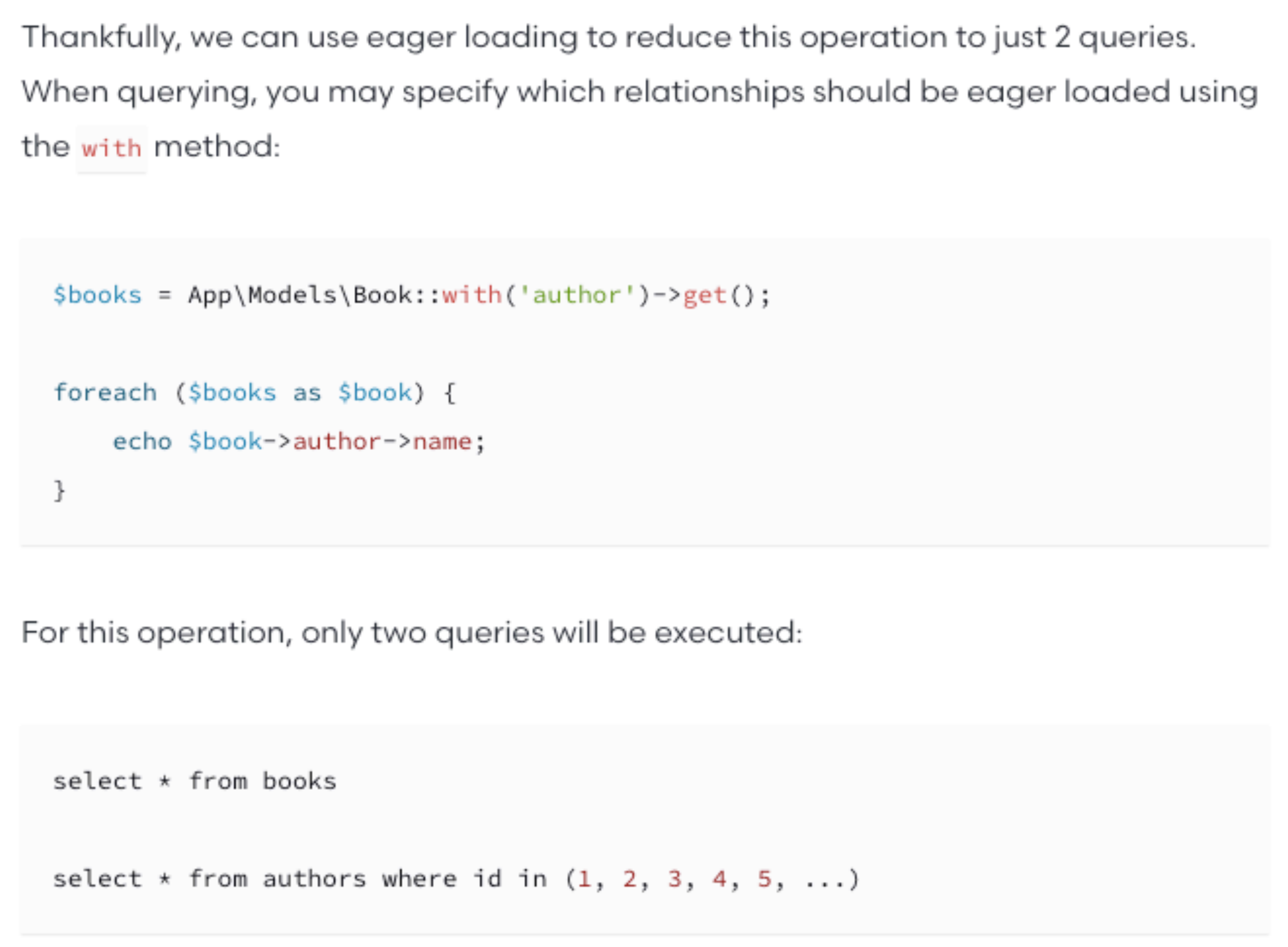
Now, let's retrieve all books and their authors:

```
$books = App\Models\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

This loop will execute 1 query to retrieve all of the books on the table, then another query for each book to retrieve the author. So, if we have 25 books, the code above would run 26 queries: 1 for the original book, and 25 additional queries to retrieve the author of each book.

PHP

ELOQUENT
Eager Loading



Thankfully, we can use eager loading to reduce this operation to just 2 queries. When querying, you may specify which relationships should be eager loaded using the `with` method:

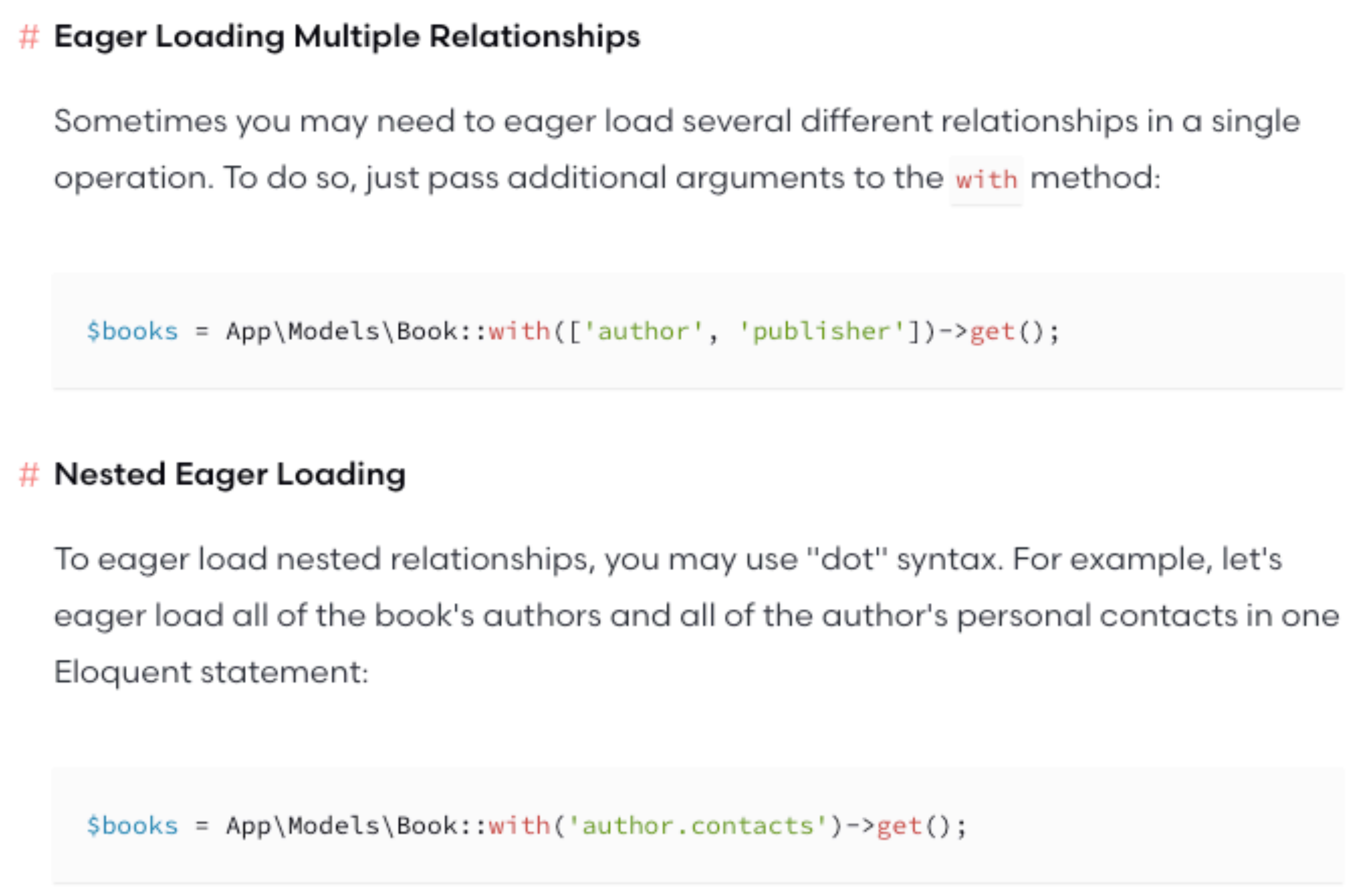
```
$books = App\Models\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

For this operation, only two queries will be executed:

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

PHP

ELOQUENT Eager Loading



Eager Loading Multiple Relationships

Sometimes you may need to eager load several different relationships in a single operation. To do so, just pass additional arguments to the `with` method:

```php
$books = App\Models\Book::with(['author', 'publisher'])->get();
```

Nested Eager Loading

To eager load nested relationships, you may use "dot" syntax. For example, let's eager load all of the book's authors and all of the author's personal contacts in one Eloquent statement:

```php
$books = App\Models\Book::with('author.contacts')->get();
```