Ralph J. Smit

Laravel Eloquent relationships for beginners - Full 2022 Guide

Laravel

Ralph J. Smit
Laravel & PHP-developer.

Laravel Eloquent is one of the flagship features of the Laravel framework. In part that is also thanks to its awesome support for defining, creating and managing relationships between different tables of data. In this tutorial I'll show you how to create and use Eloquent relationships, so that you can get up and running without any previous knowledge of relationships.

What are database relationships?

First, let's start with the absolute basics. What is a **relationship** exactly? A **relationship means that you have two or more tables with records that are related to each other**. Say that you have a `` `users` `` table, and each user could have multiple `` `posts` ``. How do you connect those two? Usually that's done by adding a `` `user_id` `` column to the `` `posts` `` table, so that you can easily identify the user that belongs to each post. This 'connection' that allows you to **figure out which records belong to each other**, is called a relationship.

What types of Eloquent relationships exist?

first and the second table.

In the above example, we have a `users` table and a `posts` table. Say that each post can only belong to **one user**. To implement this, we can add a `user_id` to the `posts` table. And in the (more unlikely) opposite scenario, if each post can belong to multiple authors (`users`) and each author can only contribute to one post, we could add a `post_id` to the `users` table. This is an example of a relatively easy kind of relationship, a **One-To-Many** relationship.

But what if each user can have multiple posts, and each post can have multiple authors. How can we solve that? This is a so-called **Many-To-Many relationship**. For this sort of complexer relationship, we can't just add a column to a table and be done with it! For this situation we need a pivot table. Don't worry if this sounds too complex, we'll get to that soon.

In general, we say that the following sorts of relationships between data exist:

1. One-To-One Relationship
2. One-To-Many Relationship
3. Has-One-Of-Many (e.g. the *latest* of many) Relationship
4. HasOneThrough and HasManyThrough Relationship
5. Many-To-Many Relationship

Do I need any specific knowledge of Eloquent before reading this?

In the below examples, I've tried to explain everything as clearly as possible, without using too much difficult Eloquent functions and complex techniques. So that means that previous knowledge is not absolutely *necessary*. However, **it is**

If you want to refine your basic Eloquent skills first, I'd recommend my previous article, which is an introduction guide to Eloquent for beginners.

One-To-One Eloquent relationship

Let's start with the **most simple relationship**: a 'Has One' or 'One-To-One' relationship. This effectively means that a certain record is related to another record (but not multiple other records!).

To **continue the blogging example** with users and posts, let's say that each User can have one Profile. In some situations, you could store all the profile information in the User model, but that wouldn't be perfect. Here, in our example, I want to store them separate table. For example, if we later want to transfer a profile to a different user, this will come in handy.

First, create a `Profile` model, the `User` model is already generated by default. The exact columns for the `Profile` model do not matter much, but you might want to **create a model ánd the migration alongside it**:

```
php artisan make:model Profile -m
```

Because this is a One-to-one relationship, we have the rare opportunity to decide if:

1. we place an `user_id` on the `Profile` model; or if
2. we add a `profile_id` on the `User` model.

something(s) else, you usually add this column to the second model. You could say that the first model 'possesses' the second model'.

So that's what I'm gonna do:

```
/** Add this to your Profile table migration or create a new migratior
$table→foreignId('user_id');
```

Run the migrations and we're ready to start implementing this.

First, open your `User` model and add the following **public function**. You are allowed to call the function whatever you like, but the convention is the `snake_case` version of the model you're referencing:

```
public function profile(): HasOne
{
    return $this→hasOne(Profile::class);
}
```

The `hasOne` **function now expects a** `user_id` **column on the** `Profile` **model**. If your column name is different, then add a second argument to the `hasOne` relationship with the other column name:

```
return $this→hasOne(Profile::class, 'author_id');
```

However, choosing something else then the convention is not recommended.

Ralph J. Smit

Adding the inverse of this relationship

In most cases we can also **define the inverse of a relationship**. Sounds complex, but it really isn't. Basically it means that we now add a function on the `Profile` model (the 'other' model; the one that has the `xxx_id` column) which points back to the model it belongs to (`User` in this case).

```
public function user(): BelongsTo
{
    return $this→belongsTo(User::class, 'author_id'); // Omit the sec
}
```

Using an Eloquent relationship

Using this **Eloquent One-To-One relationship** is very simple. Whenever you have an instance of the `User` model, you can call `$user->profile` or `$user->profile()->chainOtherMethodsHere()`.

This is because `$user` is an instance of the `User` model, and right on that model we added a `profile()` method.

Please note, and this is super important, that with this you can access both `->profile` as if it were a property and `->profile()` as if it were a function. The difference is that `->profile` returns just an Eloquent instance, whereas `->profile()` allows you to add methods to it. E.g. `->profile()->orderBy('xxx', 'ASC')->get()`.

```
$user = auth()->user();

$profile = $user->profile;
$name = $user->profile->display_name;

// Create a profile
$profile = $user->profile()->create([
   //
]);



// And:
$user = Profile::find(1)->user;

// You can use all the regular Eloquent functions on the profile() fur
// E.g. get(), where(), first() and others. */
```

Eloquent One-To-Many relationship

Another very important, perhaps even the most important one, is the **One-To-Many relationship**. Also known as the **hasMany-relationsh**ip, this relationship defines a relation that **'one items has many other items**. This relationship is very similar to the above one.

To continue our blogging example, say that a profile **has many** posts. Open your `` `Profile` `` model and add the following method:

```
{
    return $this->hasMany(Post::class);
    //Or: return $this->hasMany(Post::class, 'foreign_key');
}
```

This in fact means that each Profile has many posts. The inverse also exists:

```
public function profile(): BelongsTo
{
    return $this->belongsTo(Profile::class);
    //Or: return $this->belongsTo(Profile::class, 'foreign_key');
}
```

Using it is very similar to what we showed above, except that this **relationship returns multiple items (as an Eloquent collection)**:

Now that you've defined the relation as a function on the model, we can again use it as a **property** (`` `->posts` ``), which returns an Eloquent *collection* here, instead of a single model. And we can also use it as a function (`` `->posts()->where('created_at', '>', now()->subDays(14))->get()` ``).

```
// $posts = Profile::find(1)->posts()->get();

foreach ($posts as post) {
    // Do something
}

$lastPost = Profile::find(1)->posts()->latest()->first();
```

Adding the latest/newest and oldest model via a relationship

Sometimes when you **define a hasMany relationship**, you might want to retrieve only the **newest** or only the **oldest model**. To accomplish this, Laravel allows you to use two handy methods, **`->latestOfMany()`** and **`->oldestOfMany()`**:

```
public function latestPost(): HasMany
{
    return $this->hasMany(Post::class)->latestOfMany();
}
```

```
public function oldestPost(): HasMany
{
    return $this->hasMany(Post::class)->oldestOfMany();
}
```

```
$latestPost = Profile::find(1)->latestPost;
```

If this sounds nice, but you need some **custom filters** or other **advanced `where()` clauses**, check out this example from the Laravel docs:

```
/**
 * Get the current pricing for the product.
 */
public function currentPricing()
{
    return $this->hasOne(Price::class)->ofMany([
        'published_at' => 'max',
        'id' => 'max',
    ], function ($query) {
        $query->where('published_at', '<', now());
    });
}
```

HasOneThrough and HasManyThrough

Now that we've seen those examples, let's go one step further. Now we want to define a relationship **through an other model**.

Consider our example above, where each `User` has one `Profile`, and where each `Profile` has many `Post`s. An example of a relationship through an other model doesn't contain a `user_id`, it only has a `profile_id`. This is a perfect example of a has many through relationship.

To define such a relationship, you can do it very similar to what we saw above. In our example, add the following to the `User` model:

```
public function posts(): HasManyThrough
{
    return $this->hasManyThrough(Post::class, Profile::class);
}
```

The first argument of this function is the **model that we want to access**, and the second argument is the **intermediate model**.

Defining a `hasOneThough()` relationship is almost exactly the same as a `hasMany()` relationship, although in this situation you need to be sure that there is only one final item:

```
public function first_login(): HasOneThrough
{
    return $this->hasOneThrough(FirstLogin::class, Profile::class);
}
```

Many-to-many relationships

`User` can have many `Profile`s, e.g. when collaborating with others, and each `Profile` can belong to multiple (many) `User`s.

How do we fix this? We cannot just add a `profile_id` on the users table, because there are **potentially multiple profiles**. And neither can we add a `user_id` on the `Profile` model, because there are **potentially multiple users** as well.

To fix this, we need a sort of **intermediary table**. This intermediary table is called a **pivot tabl**e, and in most cases it serves no other function than to store two `id`s in the same row.

Consider the following scenario: if our **pivot tabl**e has a `user_id` and a `profile_id` column, we can now connect a certain `User` with a certain `Profile`. And because we can have the same `user_id` as many times as we like, we can connect each user with a many profiles as we like too. And of course this also applies to profiles: we can have as many duplicate `profile_id`s as we want, with as many different `user_id`s as we want.

How to create a migration for a Laravel pivot table?

First, let's **create the database migration** that we need. Run the following command. Notice how the table name is constructed by **joining the table names** (both singular!) in alphabetical order. You could override this, but usually there's no point in doing so.

```
php artisan make:migration create_profile_user_table --create=profile_
```

Now, open up the **migration** and **add two additional lines** to the `up()` method.

```
<?php
```

```php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateProfileUserTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('profile_user', function (Blueprint $table) {
            $table->id();
            $table->timestamps();

            $table->foreignId('profile_id');
            $table->foreignId('user_id');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('profile_user');
```

As you see, the **structure of a pivot table** is very simple. Now, run the migrations.

Next, let's open up our two models and **add the two relationship functions**. The methods on each Model are both of the form `belongsToMany()`.

Add this to your `User` model:

```
/**
 * Public function to get all the profiles this user can publish under
 */
public function profiles(): BelongsToMany
{
    return $this->belongsToMany(Profile::class);
}
```

And this to your `Profile` model:

```
/**
 * Public function to get all the users that can publish under this pr
 */
public function users(): BelongsToMany
{
    return $this->belongsToMany(User::class);
}
```

```
return $this->belongsToMany(Profile::class, 'user_role'); // The table
```

By default, **Eloquent assumes the `user_id` and `profile_id` columns on the pivot table**. If you choose to name them differently, you can use that as a third and fourth parameter:

```
return $this->belongsToMany(Profile::class, 'user_role', 'name_of_colu

// That looks unnecessarily complex, so here's an example.
// Say that we add this on the user model, by default it would look li
return $this->belongsToMany(Profile::class, 'user_role', 'user_id', 'p
```

Using an Eloquent Many-To-Many relationship

Using this relationship is **really easy** and **goes similar** to what we saw above. Checkout these small examples:

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

$user->profiles->each(function ($item) use ($user) {
    $profile->name = $user->firstname . ' ' . $user->lastname;
    $profile->save();
});
```

```
$profile→users()→orderBy('created_at', 'desc');
```

Storing data in the pivot table

The last part of this article is about **storing data in the pivot table**. Yes, that's also possible! It might be unnecessary for every use case, but there certainly are cases where this might come in handy.

As a short recap, **what is a row in a pivot table exactly**? What does it represent? The answer is that a row in a pivot table represents the **relationship between two records**. If we were to delete a specific row, would the relationship still exist? No!

Now here's why this matters. If every row in a pivot table represents a relationship, we can **also store information about the relationship on that pivot table**. This could be (relatively) trivial information about when the relation was created or last updated (with `` `created_at` `` or `` `updated_at` `` columns).

Using a pivot table to retrieve relationship data

In our above example, we **gave a migration for the pivot table**. This migration already contained the code for our `` `created_at` `` and `` `updated_at` `` timestamps (with the `` `$table->timestamps();` ``). So, how do we access this relationship?

First, we need to tell Eloquent: **our pivot model also has a few other attributes that I want to access, here they are**'.

You can do this like this:

```
     * Public function to get all the profiles this user can publish under
     */
    public function profiles(): BelongsToMany
    {
        return $this→belongsToMany(Profile::class)→withPivot('created_at
    }
```

Or use the **nice helper method `->withTimestamps()`** to already define the `created_at` and `updated_at` columns:

```
    /**
     * Public function to get all the profiles this user can publish under
     */
    public function profiles(): BelongsToMany
    {
        return $this→belongsToMany(Profile::class)→withTimestamps()→wit
    }
```

Now you can **access the pivot table** like this:

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

foreach ($user→profiles as $profile) {
    echo $profile→pivot→created_at;
}
```

Renaming the name of your pivot table

There's also one **additional nice trick you can use**, and that is **renaming the attribute `pivot`**. Eloquent is all about using descriptive and expressive language. The docs give the example of a relationship between a podcast and a user. This relationship is a subscription, so it would be a bit weird to use 'pivot' to reference to a subscription.

```
// Not nice:
$user->podcasts()->first()->pivot->price;

// Better:
$user->podcasts()->first()->subscription->price;
```

To **rename the pivot attribute**, you can use the `->as($name)` function on the `return $this->belongsToMany()` call:

```
return $this->belongsToMany(Podcast::class)
                ->as('subscription')
                ->withTimestamps();
```

Polymorphic relationships & conclusion

Wow, that was a long article! If you're reading this, thanks for joining me! In this article, I've told you a lot about all the different **kinds of relationships between Eloquent**

The relationships discussed in this article are the ones you'll use the most. And in my opinion, the above relationships are more than enough for now.

Nevertheless, you might encounter situations **where these relationships are not good enough**. Here, we always talked about relating one model to another model. But consider the relatively common situation, where you can (for example) tag both a `Post` and a `Page`. Now, the above relationships will fall short and where **polymorphic relationships** come into view. But that's a topic for another time ;-)

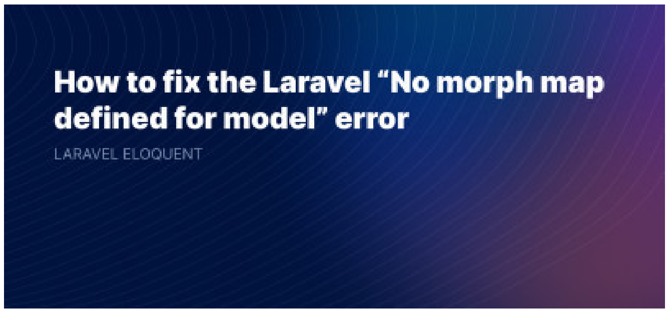
As always, if you have a question or something else, feel free to leave a comment below.

Published by Ralph J. Smit↗ on 05 October 2021 in Laravel↗. Last updated on 01 August 2022.

ralphjsmit.com/laravel-eloquent-relationships

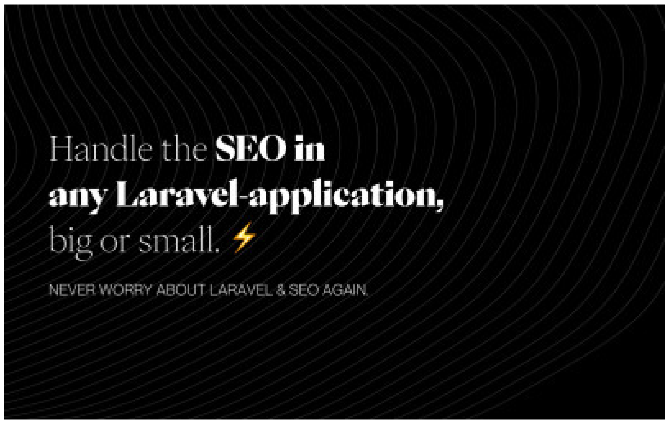
Read more

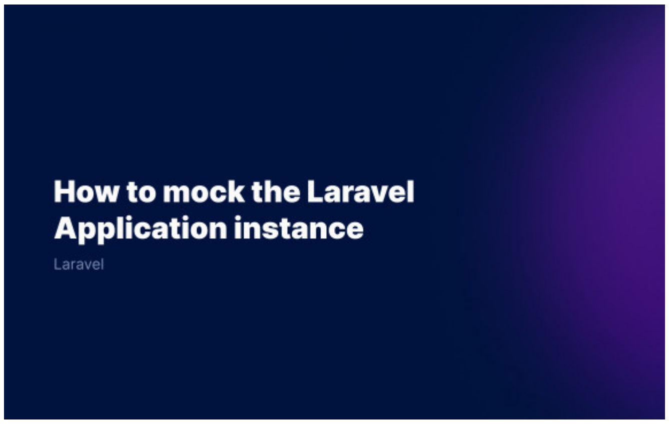
Ralph J. Smit


How to fix the Laravel "No morph map defined for model" error
LARAVEL ELOQUENT

How to fix the Laravel "No morph map defined for model" error

How to send e-mails in Laravel with Tailwind CSS


Handle the SEO in any Laravel-application, big or small. ⚡
NEVER WORRY ABOUT LARAVEL & SEO AGAIN.



New package: A package to handle the SEO in any Laravel-application

How to mock the Laravel Application instance