Pivot tables and many-to-many relationships

September 29, 2015

Today I want to talk about a feature of Laravel which is really useful but can be potentially difficult to understand at first. Pivot table is an example of intermediate table with relationships between two other "main" tables.

Real-life example of pivot tables

In official documentation they show the example of User-Role relationships, where user potentially can belong to several roles, and vice versa. So to make things clearer – let's take another real-life example: **Shops** and **Products**.

Let's say a company has a dozen of Shops all over city/country and a variety of products, and they want to store the information about which Product is sold in which Shop. It's a perfect example of many-to-many relationship: one product can belong to several shops, and one shop can have multiple products.

So here's a potential database structure:
shops
– id
– name

products
– id
– name

product_shop
– product_id
– shop_id

The final table in the list – **product_shop** is called a "pivot" table, as mentioned in the topic title. Now, there are several things to mention here.

- **Name of the pivot table** should consist of **singular** names of both tables, separated by undescore symbole and these names should be arranged in **alphabetical** order, so we have to have **product_shop**, not **shop_product**.
- **To create a pivot table** we can create a simple migration with **artisan make:migration** or use Jeffrey Way's package Laravel 5 Generators Extended where we have a command **artisan make:migration:pivot**.
- **Pivot table fields:** by default, there should be only two fields – foreign key to each of the tables, in our case **product_id** and **shop_id**. You can add more fields if you want, then you need to add them to relationship assignment – we will discuss that later.

Models for Many-to-Many Relationships: BelongsToMany

Ok, we have DB tables and migrations, now let's create models for them. The main part here is to assign a many-to-many relationship – it can be done from either of "main" tables models.

So, option 1:
app/Shop.php:

```
class Shop extends Model
{
    /**
     * The products that belong to the shop.
     */
    public function products()
    {
        return $this->belongsToMany('App\Product');
    }
}
```

Or option 2:
app/Product.php:

```
class Product extends Model
{
    /**
     * The shops that belong to the product.
     */
    public function shops()
    {
        return $this->belongsToMany('App\Shop');
    }
}
```

Actually, you can do both – it depends on how will you actuall use the relationship in other parts of the code: will you need **$shop->products** or more likely to query **$product->shops**, or both.

Now, with such declaration of relationships Laravel "assumes" that pivot table name obeys the rules and is **product_shop**. But, if it's actually different (for example, it's plural), you can provide it as a second parameter:

```
public function products()
{
    return $this->belongsToMany('App\Product', 'products_shops');
}
```

Moreover, you can specify the actual field names of that pivot table, if they are different than default **product_id** and **shop_id**. Then just add two more parameters – first, the current model field, and then the field of the model being joined:

```
public function products()
{
    return $this->belongsToMany('App\Product', 'products_shops',
      'shops_id', 'products_id');
}
```

One of the main benefits here: **you don't need to create a separate model for ProductShop** – you will be able to manage that table through pivot commands, we will discuss that right away.

Managing Many-to-Many Relationships: attach-detach-sync

So, we have tables, and we have Models ready. Now, how do we actually save the data with a help of our two Models instead of the third intermediate one? Couple of things here.

For example, if we want to add another **product** to the current **shop** instance, we use relationship function and then method **attach()**:

```
$shop = Shop::find($shop_id);
$shop->products()->attach($product_id);
```

The result – a new row will be added to **product_shop** table, with **$product_id** and **$shop_id** values.

Likewise, we can **detach** a relationship – let's say, we want to remove a product from the shop:

```
$shop->products()->detach($product_id);
```

Or, more brutally, remove all products from a particular shop – then just call method without parameters:

```
$shop->products()->detach();
```

You can also attach and detach rows, passing array of values as parameters:

```
$shop->products()->attach([123, 456, 789]);
$shop->products()->detach([321, 654, 987]);
```

And another REALLY useful function, in my experience, is updating the whole pivot table. Really often example – in your admin area there are checkboxes for shops for a particular product, and on **Update** operation you actually have to check all shops, delete those which are not in new checkbox array, and then add/update existing ones. Pain in the neck.

Not anymore – there's a method called **sync()** which accept new values as parameters array, and then takes care of all that "dirty work" of syncing:

```
$product->shops()->sync([1, 2, 3]);
```

Result – no matter what values were in **product_shop** table before, after this call there will be only three rows with shop_id equals 1, 2, or 3.

Additional Columns in Pivot Tables

As I mentioned above, it's pretty likely that you would want more fields in that pivot tables. In our example it would make sense to save the **amount of products**, **price** in that particular shop and **timestamps**. We can add the fields through migration files, as usual, but for proper usage in relationships we have to make some additional changes to Models:

```
public function products()
{
    return $this->belongsToMany('App\Product')
        ->withPivot('products_amount', 'price')
        ->withTimestamps();
}
```

As you can see, we can add timestamps with a simple method **withTimestamps** and additional fields are added just as parameters in method **withPivot**.

Now, what it gives us is possibility to get those values in our loops in the code. With a property called **pivot**:

```
foreach ($shop->products as $product)
{
    echo $product->pivot->price;
}
```

Basically, **->pivot** represents that intermediate pivot table, and with this we can access any of our described fields, like **created_at**, for example.

Now, how to add those values when calling **attach()**? The method accept another parameter as array, so you can specify all additional fields there:

```
$shop->products()->attach(1, ['products_amount' => 100, 'price' => 49.99]);
```

Conclusion

So, pivot tables and many-to-many relationships are handled quite conveniently with Eloquent, so there's no need to create a separate model for intermediate table. Hope that helps!

Want to learn more?

Watch my free video called Advanced Pivot Tables in Many-to-Many.

It's one preview lesson from my online-course Eloquent: Expert Level.

Povilas Korop

PHP web-developer with 15 years experience, 5 years with Laravel. Now leading a small team of developers, growing Laravel adminpanel generator QuickAdminPanel and publishing Laravel courses on Teachable.