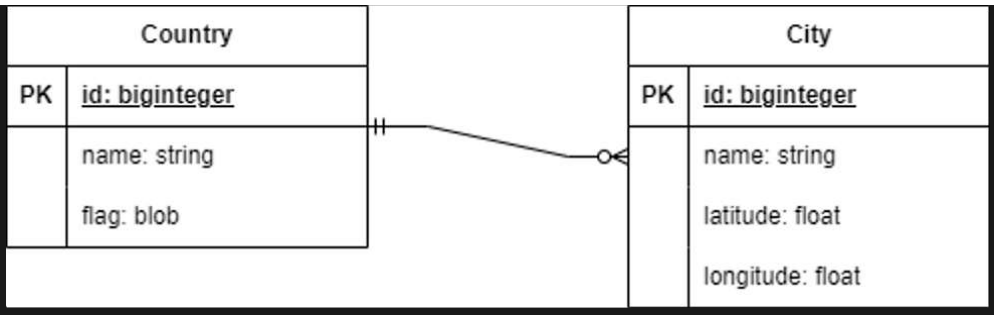
Eloquent relationships

The essentials

Relating data is one of the most important concepts of relational databases. And it's obvious that Laravel's Eloquent ORM supports relationships in a fluent OO-style manner. Fetching the collection of cities that belong to a country for example can be done by just calling a method on the parent country model object.

There are different types of relationships, like one-to-many and many-to-many. This chapter will explain how Laravel supports the one-to-many relationship, since it is the most used one. We will use the following example:



Conceptual model of a one-to-many relationship

- Link to Databases course
- We stick to one-to-many ones (mandatory and optional)

This means that countries might contain (zero or more) cities and that a city always belongs to a country.

> ⚠️ Note that the essentials section is assuming a mandatory one-to-many relationship. This means in the example that a country must belong to a city. It also assumes default referential actions to be default to the database system (which is RESTRICT when using MySQL). This means that an error will be generated if you want to delete a country that has related cities. Check the Digging Deeper section if you need other settings.

Naming conventions

As usual, Laravel makes things easier here if you follow conventions. In this case, it is mostly about naming:

- Each table is named as per the Laravel conventions: lower case plural form of the Model name (i.e. User → users, Country → countries and City → cities)
- Each table is expected to have the default Laravel primary key (`id` column)
- The foreign key column is named as the "snake case" name of the parent model followed by `_id`. In the Country → City relationship for example, the foreign key column in the cities table that references the country where the city belongs to is named `country_id`

Defining relationships in Migrations

Defining a relationship in a database means you need to do two things:

1. Add a Foreign Key column to the table that holds the 'many' entities. In our example the City. The cities table needs a column to store the id of the country where the city belongs to.

2. Add a special index called a *Foreign Key Constraint* which help keep the related data consistent

Laravel supports this in one command (assuming the Laravel conventions are met) for your migrations:

```
Schema::table('cities', function (Blueprint $table) { $table->id(); // ... $table->foreignId('country_id')->constrained(); // ... $table->timestamps(); });
```

Defining one-to-many relationships

The next step is to extend both models with methods and properties such that the related data can be fetched and managed.

Defining the relationship in the parent model

Laravel assumes this to be the default definition of the relationship (the other way around is called 'inverse'). Like all other Eloquent relationships, one-to-many relationships are defined by defining a method on your Eloquent model:

```
<?php namespace App\Models; use Illuminate\Database\Eloquent\Model; use Illuminate\Database\Eloquent\Relations\HasMany; class Country extends Model { // ... /** * Get the cities for the country. */ public function cities(): HasMany { return $this->hasMany(City::class); } }
```

Remember that this will only work like so when the Laravel conventions are met. The method returns actually an object which is like a Query, which you can extend with other Query builder methods. It also adds a "magic property" `cities` to the model which is filled with the resultset of the query. We'll address this when using the relationship.

The inverse relationship: belongs to

To define the inverse of a `hasMany` relationship, define a relationship method on the child model which calls the `belongsTo` method:

```
<?php namespace App\Models; use Illuminate\Database\Eloquent\Model; use Illuminate\Database\Eloquent\Relations\BelongsTo; class City extends Model { // ... /** * Get the country where the city belongs to. */ public function country(): BelongsTo { return $this->belongsTo(Country::class); } }
```

Just like `HasMany` the method returns a query-like object and adds a magic attribute `country` that holds the parent model.

Using the relationship

Using a relationship means that you can read and modify related data. You might want to render all the cities from a certain country in a details view of that country or add a new city to that country for example. Most of this work can be done easily in an OO manner by using the `HasMany` and `BelongsTo` definitions in the Eloquent models.

Reading related data

The simplest way of reading all the related data is by using the magic attributes that the `HasMany` and `BelongsTo` definitions added to the models. To read all the cities that are related to a country for example is done like so:

```
$country = Country::find(1); $cities = $country->cities; // $cities is a collection of City objects
```

And for the inverse relationship:

```
$city - City::find(21); $country = $city->country; // $country is a Country object
```

Inserting & Updating Related Models

You can also use the `HasMany` and `BelongsTo` definitions for inserting and updating related models. When for example you want ta add a new city to a country, you can use the `save()` method:

```
$country = Country::find(1); $city = new City(); $city->name = "Middelburg"; $city->latitude = 51.5; $city->longitude = 3.61389; $country->cities()->save($city); // Sets the city FK attribute Saves the saves to the DB
```

Or, when you want to utilize mass assignment, the `create()` method:

```
$country = Country::find(1); $city = $country->cities()->create([ "name" => "Middelburg", "latitude" => 51.5, "longitude" => 3.61389 ]); // Create a city, with the FK attribute set in the DB
```

Note the expressiveness: you are creating a new city within a country's cities.

In the inverse relationship, you can use the `associate()` method:

```
$city - City::find(21); $country = Country::find(1); $city->country()->associate($country); // Updates the FK column $city->save();
```

Optional relationships

Sometimes, relationships are optional. a Task *might* belong to a project or a musician might be member of a band. In a one-to-many relationship, this only means that the foreign key column does not require to have a value. If you want to implement this using a Laravel migration, you can simply add a `nullable()` modifier like so:

```
Schema::table('musicians', function (Blueprint $table) { $table->id(); // ... $table->foreignId('band_id')->nullable()->constrained(); // ... $table->timestamps(); });
```

When using the relationship you must always keep in mind that the BelongsTo magic attribute `$musician->band` might return `null` if the musician does not belong to any band. It also means that you must be able to set the foreign key column to `null`, which can be done with the `dissociate()` method:

```
$eric = Musician::where('name', '=', 'Eric Clapton')->get(); // Eric left Derek and the Dominos and went on solo $eric->band()->dissociate();
```

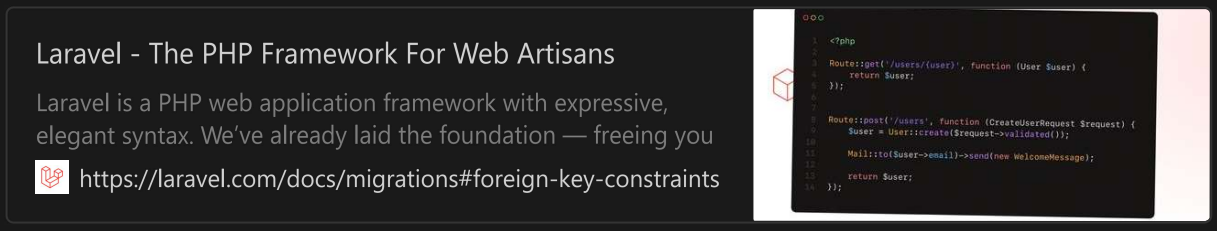
Digging deeper

Setting referential actions

MySQL defaults referential actions to RESTRICT. If you want to change this, you need to modify the `foreignId()` declaration in your migration. The following example shows how you can set it to CASCADE:

```
Schema::table('cities', function (Blueprint $table) { $table->id(); // ... $table->foreignId('country_id') ->constrained(); ->cascadeOnUpdate() ->cascadeOnDelete(); // ... $table->timestamps(); });
```

If you want to see other options, check the next section of the Laravel Migrations documentation:



Using the HasMany query

It is also good to realize that all relationships also serve as query builders. This means that you can also add more filtering, grouping and ordering, for example:

```
$country->cities() ->where('name', 'like', '%dam') ->orderBy('name', 'desc') ->get();
```

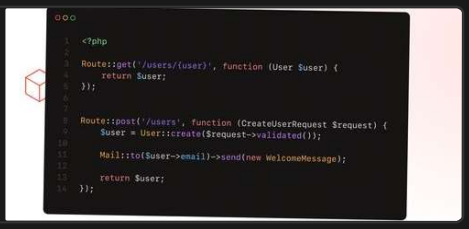
Further reading

If you want to read more about Relationships in Laravel, here is a link to its documentation site on this topic:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/eloquent-relationships



You can read for example about:

- Other types of relationships like one-to-one and many-to-many
- Polymorphic relationships, which is a bit like inheritance