A Laravel Prime (Part 2)

Using a Database

Most useful applications will require manipulating data in several ways. The most common source of this data is a Database, so it is no surprise that you will find yourself wanting to use Laravel and a Database at the same time. Also, as you know, the most commonly used database in the web is probably MySQL, so that's what we are going to be using. Laravel however provides you with ways of using other databases, as we will see in a few moments.

Task 1: Set-up

The first thing you need is a database to connect to. This is the only step that needs to be performed outside of Laravel. Log-in to your database server using your preferred method, for example by navigating to http://localhost/phpmyadmin
Create a new user for your project, and then create a database for that user, granting all privileges to this user for the database of your project ONLY. This is considered good practice.

While it is possible to manually create the database tables using a separate tool, like phpMyAdmin, or SQL Workbench or similar tools, Laravel provides you with a way to manipulate the database without writing a single line of SQL, by fully utilising the Eloquent ROM (Relational Object Model).

The first step however requires setting up the connection details to your database. This is done from the application config folder. In there you will find a database.php file which sets up some default values. On a fresh installation of Laravel, the default connection will be set as 'mysql'. This is fine for us as we will be using the MySQL server. Further down in the same file, an array 'connections' sets the default connection values to several different databases. One of them is the mysql, which sets up the host, database, username and password details for this database.

Task 2: configuring the conection

In the .env file at the root of your app, modify the values for the database, username and password to match those used in Task 1 above. You can leave the host as 'localhost'.

Laravel is now ready to use your database. But how do you start creating tables and adding data to the tables?

Enter Migrations

You can think of migrations as a form of version control for your databases. Migrations are very powerful and perhaps you won't appreciate the true power of them until you start working on your first full-fledged project. However it is a very powerful tool to learn from the start. Migrations are part of Laravel and are run from the command line. In order to use them, you first need to "install" the migration table on your database. This is done with a simple command

Task 3: Install the migrations table

Open a command line window or terminal, and in your project folder run the command:

```
php artisan migrate:install ↵
```

Then verify in phpmyadmin that the migration table has been created inside the project database.

This last step serves two purposes: it creates the table that will be used to keep track of the order of your migrations, allowing you to rollback changes, and tests that the connection to the database is working as intended.

Migrations are a programmatical way of manipulating the structure of your database. Because of their programmatical nature, they provide superb functionality and flexibility. Plus it removes the need to know any SQL at all, although it doesn't hurt to know some.

Laravel ships with a couple of migrations by default. You can examine those to get a feeling of how they are created. There's nothing stopping you from creating your migrations manually, but Artisan can simplify the task for you.

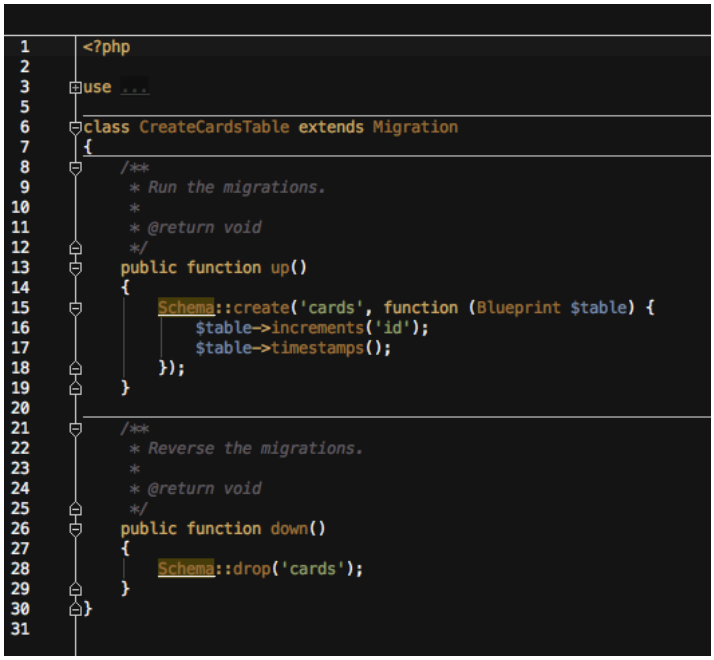
Creating a table with a migration is extremely simple. You can do so from the command line. Let's start creating a Cards table with migrations:

Task 4: your first migration

Still in the terminal/command line window, type the following command:

```
php artisan make:migration create_cards_table  --create=cards ↵
```

Because we have used the optional flag `--create=cards` artisan has understood your migration intends to create a new table, called cards and has gone ahead and given you some boilerplate code to get you started. The migration file can be found in the database/migrations folder under your app directory. Open it now; you should see code similar to this:



```php
<?php

use ...

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

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

This class CreateCardsTable has two methods: up() and down(). Up() will be called when you apply the migration. In this method you want to initialise the structure of your table. Laravel has given you a start, by creating a primary key 'id' and some timestamps that will be useful for the future. If you ever need to reverse this particular migration, the down() method would be called. Because you would need to drop the table completely in order to undo whatever changes you perform to your database here, Laravel has already provided you with the code to do that in the down() method.

Currently this table only would have an id column, and two timestamps column indicating when the row was created (created_at) and when it was last modified(updated_at). This is no use for our Cards table. We need some more rows. Your cards table would likely require a title, and for now that will be enough. You can define a title by adding the line

```
$table->string('title');
```

to the up() function, just after the id. The result as you will find is a field of VARCHAR type with a default value of 255 chars.

For now, this is all we will need. Once a migration has been created, you can execute it by running the command

```
php artisan migrate ↵
```

This will execute the up() method of your migration. If you ever need to rollback your migration, you can issue the command

```
php artisan migrate:rollback ↵
```

which will execute the down method of your last migration. You can continue to issue this command to roll back sequentially through your migrations. You can also rollback all your migrations by using `migrate:reset` as the last part of the command. If you want to rollback all your migrations and then run them all again (for example, to apply all the changes from fresh), you can use `migrate:refresh`. Notice that when you run the command, Laravel migrates two more tables, which are shipping with Laravel by default: Users and Password Resets. We will not be using those yet, nevertheless they are useful things, so we will leave them for the time being.

Tinkering with Artisan

Here's a command that will let you interact with your application:

```
php artisan tinker ↵
```

Tinker will let you issue some commands and see the immediate result in an interactive fashion. You can issue any command like

```
2 + 2
```

and tinker will provide you with an appropriate output. You can also use php functions, but more importantly, you can interact with your application. For example you can interact with the recently created table in your database. Try this to add a record to the database:

```
DB::table('cards')->insert(['title'=>'My New Card', 'created_at'=>new
DateTime, 'updated_at'=>new DateTime]); ↵
```

Let's see the row we just created:

```
DB::table('cards')->get(); ↵
```

We can also run something akin to SQL for filtering data, like

```
DB::table('cards')->where('title','My New Card')->first();↵
```

Let's add another card. You should be able to hit up a few times and cycle through your recent commands.

```
DB::table('cards')->insert(['title'=>'My Second New Card',
'created_at'=>new DateTime, 'updated_at'=>new DateTime]); ↵
```

So we have created a table structure in a database through a migration, and we have added some data to it programmatically, and so far we haven't written a single line of SQL code. How do we output some of this data to the browser using Laravel?

We haven't actually done much yet in terms of using models. Laravel ships with one for users but we haven't made use of it. Still, this has allowed us to test the

waters, get our toes wet with this new framework. If you have been following from the previous document, you should have a general idea how routes work and how controllers and views work. Let's put the final ingredient in the mix, the models. We want to create a page that will list all the cards in our cards table (why? I don't know, just as an exercise!), so go ahead and create your route, controller and view for this. Just keep it simple, create a generic view that outputs a message such as "Here goes all cards", we will populate it in a moment. Just to refresh your memory, I will list the steps you should follow:

- Create a route to your '/cards' page pointing to the CardsController@index method
- Create the 'CardsController.php' file/class, which extends the base Controller class (in the Controllers folder):
  `php artisan make:controller CardsController ↵`
- Create the index() method in the CardsController class, returning a view to an index page, perhaps in the views/cards folder
- Create the view for the CardsController@index method (in the views/cards folder).

When you point your browser to the http://localhost:8000/user page, it should take you to your recently created index view.

Let's start with using the DB class directly to manipulate the database. This is called the Query Builder in Laravel. Modify the CardsController file so that it looks like this:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;
use DB;

class CardsController extends Controller
{
    public function index()
    {
        $cards = DB::table('cards')->get();

        return view('cards.index')->withCards($cards);
    }
}
```

and your view should look similar to:

```
@extends('layouts.main')

@section('content')

    <h1>All Cards</h1>

    @foreach ($cards as $card)
        <div>
            {{$card->title}}
        </div>
    @endforeach

@stop
```

Test this in your browser.

You need to be Eloquent to talk to your database

When you point your browser to the http://localhost:8000/cards page, it should take you to your recently created index view. We want to display the titles of all cards in this page. Our controller must get the data from somewhere, so we can do that by using the Query Builder and adding this familiar line before our return in the controller:

```
$cards=DB::table('cards')->get();
```

Of course, our controller will want to pass this information to our view, so in our return line, we want to add the data. We know few ways of doing this. Here's my favourite:

```
return View::make('cards.index')->withCards($cards);
```

Next, our View must be able to show this data. First of all, I want you to try just outputting the data "raw". What does it look like? In your view file, echo the $cards variable:

```
{{ $cards }}
```

refresh your web browser:


```
[
  ▾ {
      id: 1,
      title: "My new card",
      created_at: "2016-02-10 20:44:48",
      updated_at: "2016-02-10 20:44:48"
    },
  ▾ {
      id: 2,
      title: "My second new card",
      created_at: "2016-02-10 20:46:57",
      updated_at: "2016-02-10 20:46:57"
    }
]
```


What you are getting is what's called a JSON formatted string. JSON stands for JavaScript Object Notation, and it's one of the most popular ways of transferring data between web applications. I'm using a safari plugin to prettify the JSON and make it more readable, you can find similar ones for Chrome and Firefox. You can identify easily the different fields of your cards table: id, title and the two timestamps created_at and updated_at. But how can you actually use these in your web app now?

Try this instead in your view:

```
All the cards:
	<ul>
	@foreach ($cards as $card)
		<li>{{ $card->title }}</li>
	@endforeach
	</ul>
```

But we are still not using models. We have made use of the Query Builder again. How about we instead delegate all the grunt work to Laravel? Let's try that. In the console, run this command to create a Model for your Cards. Note the convention for naming the model, it should use the singular form of your table name:

```
php artisan make:model Card ↵
```

Now, in your controller, replace the line

```
$cards=DB::table('cards')->get();
```

with

```
$cards=Card::all();
```

Make sure you are making use of the Card model instead of the DB class by changing the line

```
use DB;
```

with

```
use App\Card;
```

And try it again in the browser.

Have you realised that you didn't have to worry about making a connection to your database, selecting a table, running a query, manipulating the result, or closing the connection to the database? Even better, you didn't even have to think about SQL at all!

Now, what about if I want to get a specific card's data? For example for the card with id 21?

Simple, in your CardsController you would use:

```
$card = Card::find(21);
```

and would adjust your view appropriately to work with a single card, rather than an array, for example:

```
<div>
      Title: {{ $card->title }} <br />
      created: {{ $card->created_at }}<br />
      last updated: {{$card->updated_at }}<br />
</div>
```

Ok, but what if I want to select a random card, perhaps from the list of all cards?

You can pass a wildcard to your controller from your route like so:

```
Route::get('cards/{card}'),'CardsController@show');
```

Then create a new method show() in your CardsController:

```
public function show(Card $card)
	{
		return View::make('cards.show')->withCard($card);
	}
```

and finally a new view show.blade.php with the appropriate code as before.

To test it, point your browser to, for example

http://localhost:8000/cards/1

Notice that by using type hinting in the show method of the CardsController, we were able to skip the process of telling Laravel to fetch that particular card. This will only work if we use appropriate names for the token (in the route) and the variable (in the controller). Otherwise we must explicitly tell the model what we want to achieve, for example:

```
public function show($id)
	{
		$card = Card::find($id);
		return View::make('cards.show')->withCard($card);
	}
```

Where possible, use type hinting as it will make your code that much more readable and maintainable.

Eloquent Relationships

So far our Cards are a bit underwhelming. They haven't got much meat to them. Presumably, each card will hold a number of notes inside. This means we will likely need a table to hold our notes and the relationship Cards:Notes is a 1:N. How do we work with this in Laravel?
Let's start with an exercise:

Task
Create a migration for the notes table. Notes consist of an id, a card_id (integer, unsigned and indexed), a body(of type text) and the relevant timestamps. And then run the migration. Also create a model for your notes called Note.

We can use tinker to create our first few notes.
Issue these commands in tinker:

```
$card=App\Card::first();
$note = new App\Note;
$note->body = 'Some Note for the card';
$note->card_id=$card->id;
$note->save();
```

But let's make use of the Models from now on.
First we need to tell Laravel about the relationships. A card will have many notes. So open the Card model and add a new method notes:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Card extends Model
{
    public function notes()
    {
        return $this->hasMany(Note::class);
    }
}

```

Notice how easy it is to associate the Card with its Notes. In clear language you are saying this has many notes.

We also need to deal with the reverse relationship. A Note belongs to a Card. In the Note model, create a new method called card:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Note extends Model
{
    public function card()
    {
        return $this->belongsTo(Card::class);
    }
}

```

Now that Laravel knows about the relationships, it should be easier to add notes to the database. Let's see this in tinker first:

```
$card=App\Card:: first();
$note = new App\Note;
$note->body='This is another note';
$card->notes()->save($note);
```

note that Laravel automatically sets up the card_id attribute for you.
Now try

```
$card->notes;
```

Now we are already displaying some information about one card with the show method and view. We want to be able to show all notes associated with a card there too. Use what you know to change the show.blade.php view to display all notes associated with the current card.

Let's try a different approach. You don't have to create the note before you associate it with the card. You can do that in one line too like so:

```
$card->notes()->create([‘body’ => ‘yet another note about this
card’]);
```

However if you try this in tinker, you should receive a “MassAssignmentException” message. What does that mean? Laravel is actually trying to protect you from possible injection attacks. It is basically saying “Someone is trying to mass assign values to the card model”. To avoid this, you need to explicitly say which values can be mass assigned (passed as members of an array). Head to the Note model and add this line as part of the Note class:

```
protected $fillable =
	[
		‘body’
	]
```

If you try it again, this time it should go through. And still not a single SQL line has been used.

Let’s finish up for now with the link from the Card index page to the Card details page.

```
<a href="cards/{{$card->id}}"> {{$card->title}}</a>
```

Try this and test your page.