Laravel Project Report

Matthew Pantaleon

Table of Contents

Laravel

First thing I did was create a new laravel application. I made sure I had composer installed and made sure that the command prompt knows where composer was I typed in `composer global require laravel/installer` to get the laravel installer. Again I made sure the command prompt knows what laravel is then I typed in `laravel new blog`. While working on different machines in the college I was not able to use laravel it self so I typed a composer comamnd that does the same thing `composer create-project --prefer-dist laravel/laravel NAME`. NAME is the name of the laravel application that will be created.

Database

Now that I have fresh laravel application I moved on to the database. The database structure consists of 6 main tables. Patients, Doctors, Visits, companies, users and roles.
Patients and doctors have a many-to-many relation ship through the visits table.
Patients can have one company and a company can have many patients. Making it a one-to-many.
Users can only have one role and a role can have multiple users. Also making it a one-to-many.

below it the structure of the database.

Migrations

With the structure of the database down I needed to create the tables. Laravel ofers migrations. Migrations are what laravel uses to be able to create database tables. I used comamnds the artisan command e.g.`php artisan make:migration create_doctors_table` to create blank migration files. I created one for each table I need. With the exception of the users table I used `php artisan make:auth` This creates all the user related migration, model, view ad controller files for user functionality. Login and register.

I edited each file to have any columns I wanted it each table and added constraints to the table where it was needed. For example the visits table.

```
<?php
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateVisitsTable extends Migration
{

    public function up()
    {
        //creates the columns of the visits table
        Schema::create('visits', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('doctor_id')->unsigned();
            $table->integer('patient_id')->unsigned();
            $table->date('date');
            $table->time('time');
            $table->integer('duration');
            $table->double('price', 15, 2);
            $table->timestamps();

            //adds foreign keys on doctors_id and patients_id on doctors and
patients table respectively
            $table->foreign('doctor_id')->references('id')->on('doctors');
            $table->foreign('patient_id')->references('id')->on('patients');
        });
    }


    public function down()
    {
        Schema::dropIfExists('visits');
    }
}
```

After migrating one a SQL an error is returned saying *specified key was too long*. This is because the xampp version that we use runs on an SQL version lower than what laravel supports by default. To fix this I went to the *AppServiceProvider* file and edited as such:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Schema; //imports the Schema class

class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        Schema::defaultStringLength(191); // fizes SQL error of key being too long
    }
}
```

Since my tables have foreign key constraints, the foreign key constraints must be unsigned integers and the order in which the migrations or executed is very important. If I migrate the visits table first before either the codtors or patients. Laravel will return an SQL error. I have a similar situation for companies and patients where companies table must be created first.

Laravel executes the migrations based on the date order of the files. As shown here the visits table is the last file to be migrated. I didn't know it did this as I made my visits migration file before my patients file after realising this then changed the date on the visits migration.

ENV

Now that I had my migrations laravel needs to know where I'm migrating the files to, to create databae tables. In the ENV file in the root of the application I made sure that the database name exists and set the username and password for phpMyAdmin.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=medical
DB_USERNAME=root
DB_PASSWORD=
```

Now Laravel knows where to migrate the files. I migrated them using the command: `php artisan migrate` and this will create empty tables in the database as well as a migrations table to keep a record of all the migrations that laravel has executed. If I found out that there was something I missed in the migrations or an error then I fix it. I can execute: `php artisan migrate:refresh`. This will rollback all existing migrations and migrate again.

Now there are empty tables in database we need data to initially populate the tables.

Models

Before wI seeded the database, I needed to create the models. Models are the way laravel converts database data into PHP objects for use the application and vice-versa save into the database. Using the command: `php artisan make:model Name` where name is the name of the blank model class that is to be created for a specific table. By default the model will assume the name of its table is the plural of its name. If you have a different table name from the one that laravel uses by default, you can set `$table` to a custom name. I set my table name to what laravel would use to be on the safe side.

While migrations deal with database creation and database constraints. Models are used to create objects and create relational queries based on the database constraints.

Below are the models for doctors, patients and visits. There are more models in my project but these three highlight the core principles of models.

Doctor Model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Doctor extends Model
{
    protected $table = 'doctors';

    public function visits(){
        return $this->hasMany('App\Visit');
    }
}
```

Patient Model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Patient extends Model
{
    protected $table = 'patients';

    public function company(){
        return $this->belongsTo('App\Company');
    }

    public function visits(){
        return $this->hasMany('App\Visit');
    }
}
```

Visit Model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Visit extends Model
{
    protected $table = 'visits';

    public function patient(){
        return $this->belongsTo('App\Patient');
    }

    public function doctor(){
        return $this->belongsTo('App\Doctor');
    }
}
```

This allows laravel to retrieve data relative from other tables. I.e. `Patient::all()->first()->visits()` this will return all visits for the first patient in the database. One-to-Many. `Visit::all()->last()->doctor()->name` this will return the doctor name for the last visit in the database. One-to-Many inverse.

I used `php artisan tinker` to test the relational queries as I went along to make sure the queries worked.

Seeders

Seeders are a way to automatically seed tables with immediate data, mainly used for testing and setup purposes.

To quickly duplicate blank seeder files I used the comamnds: `php artisan make:seeder UsersTableSeeder`, `php artisan make:seeder DoctorsTableSeeder`. etc... for each model class.

With the models defined putting data in the database becomes easier as I can just create each table's repective model objects and save them.

For example the *VisitsTableSeeder.php*:

```
<?php

use Illuminate\Database\Seeder;
use App\Role;
use App\User;
use App\Doctor;
use App\Visit;
use App\Patient;

class VisitsTableSeeder extends Seeder
{
    /*      Table Reference

            $table->increments('id');
            $table->integer('doctor_id')->unsigned();
            $table->integer('patient_id')->unsigned();
            $table->date('date');
            $table->time('time');
            $table->integer('duration');
            $table->double('price', 15, 2);
            $table->timestamps();
     */

    public function run()
    {
        $visit1 = new Visit();
        $visit1->doctor_id = Doctor::where('email', 'Al@Gore.com')->first()->id;
        $visit1->patient_id = Patient::where('email',
'Kyle@Reed.com')->first()->id;
        $visit1->date = '2017-05-20';
        $visit1->time = '12:00';
        $visit1->duration = 30;
        $visit1->price = 65.00;
        $visit1->save();
        $visit1 = new Visit();
        $visit1->doctor_id = Doctor::where('email', 'Al@Gore.com')->first()->id;
        $visit1->patient_id = Patient::where('email',
'danny@phantom.com')->first()->id;
        $visit1->date = '2017-05-22';
        $visit1->time = '12:00';
        $visit1->duration = 30;
        $visit1->price = 65.00;
        $visit1->save();

        $visit1 = new Visit();
        $visit1->doctor_id = Doctor::where('email', 'dr@suess.com')->first()->id;
        $visit1->patient_id = Patient::where('email',
'carl@boone.com')->first()->id;
        $visit1->date = '2017-05-22';
        $visit1->time = '12:00';
        $visit1->duration = 30;
        $visit1->price = 65.00;
        $visit1->save();

    }
}
```

To seed the tables I used the command: `php artisan db:seed`. Migrations used date stamps in their name to determine execute order but the seed command goes to *DatabaseSeeder.php* and executes each seeder file line by line, so it is important that tables that have constraints on other tables be seeded last to ensure integrity. Using the Model's ability to do relational queries, seeder values that need values from other tables are never fully hardcoded. Further enforcing foreign key constraints.

Here Users belongs to Roles. So Roles are seeded first, Patients belong to Companies and Visits belongs to Doctors and Patients and is seeded last.

```
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{

    public function run()
    {
        $this->call(RolesTableSeeder::class);
        $this->call(UsersTableSeeder::class);
        $this->call(DoctorsTableSeeder::class);
        $this->call(CompaniesTableSeeder::class);
        $this->call(PatientsTableSeeder::class);
        $this->call(VisitsTableSeeder::class);
    }
}
```

Once I have fully checked that the Model realtionships are working. The front end of the application now gets done.

Routes

Routes are how users navigate through your site, however many pages you make it. Since my project has roles, routes are the first place where these roles really show on what can be accessed. I have an admin and user role I made it so that admins can create, read, update and delete all three types. Doctors, patients and visits. While users can CRUD visits but only read doctors and patients.

```php
<?php

Route::get('/', function () {
    return view('welcome');
});

Auth::routes();

Route::get('/home', 'HomeController@index')->name('home');
Route::get('admin/home', 'HomeController@index')->name('admin.home');
Route::get('user/home', 'HomeController@index')->name('user.home');

Route::resource('admin/doctors', 'admin\DoctorController', array("as" =>
"admin"))->middleware('auth');
Route::resource('admin/patients', 'admin\PatientController', array("as" =>
"admin"))->middleware('auth');
Route::resource('admin/visits', 'admin\VisitController', array("as" =>
"admin"))->middleware('auth');
//Route::resource('user/doctors', 'user\DoctorController', array("as" => "user"));

//Route::resource('/doctors', 'user\DoctorController', array("as" =>"user"));
Route::get('/doctors',
'user\DoctorController@index')->name('user.doctors.index')->middleware('auth');
Route::get('/doctors/{doctor}',
'user\DoctorController@show')->name('user.doctors.show')->middleware('auth');

//Route::resource('/patients', 'user\PatientController', array("as" =>"user"));
Route::get('/patients',
'user\PatientController@index')->name('user.patients.index')->middleware('auth');
Route::get('/patients/{patient}',
'user\PatientController@show')->name('user.patients.show')->middleware('auth');

Route::resource('/visits', 'user\VisitController', array("as" =>"user"));
```

`Route::resource(/BaseUrl, Controller, array("as" => prefix))` Routes to all seven default functions to which ever controller you route it to. It takes a base url to build upon more urls from it, the controller to route to and a name prefix that will be applied to all routes for this url.

Admins have the resource route for doctors, patients and visits. As for users I manually defined index and show for doctors and patients. With this I checked whether or not it was working. I made the controllers using the comamnd: `php artisan make:controller ControllerName --resource` for resource controllers with the functions already made and `php artisan make:controller ControllerName` for blank controllers without the prebuilt fucntions.

I put the controllers in the correct file paths and made sure that the name space of the controllers are correct and made sure they can still inherit from the Controller Super Class so they still work.

I tested if the controllers were being used by returning a string confirming the routes do work.

Controllers

At this point the controllers don't really do anything, they just print out a string to the page.

Index and show

I started with displaying all the doctors for both admins and users. All it does it return all the doctors and passes the array to the respective view. View for patients and visits are the same just with a different view path and array values. Show is nearly identical to index but it only passes one object to the respective show view.

I have it so that when viewing a single doctor all visits for that doctor are shown as well, I pass in patients as well so as to display the patient names for those visits so it makes sense to the user. Similar idea for patients but vists do not have to ability to be displayed as a single object.

```php
<?php

public function index()
    {
        $doctors = Doctor::all();
        return view('admin/doctors')->with('doctors', $doctors);
    }

public function show($id)
    {
        $doctor = Doctor::findOrFail($id);
        $visits = Visit::all();
        $patients = Patient::all();

        return view('admin/showDoctor')->with(['doctor' => $doctor, 'visits' =>
$visits, 'patients' => $patients]);
    }
```

Create and Edit

Create and Edit functions in the controller just returns the views for the create and edit forms for doctors, patients and visits for admins and visits for users. Key difference being the edit form is being passed the appropriate object that is to be edited.

```php
<?php

public function create()
    {
        return view('admin/doctors/create');
    }

public function edit($id)
    {
        $doctor = Doctor::findOrFail($id);
        return view('admin/doctors/edit')->with('doctor', $doctor);
    }
```

Store and update

Store and update are routed to from the create and edit view forms. This is where the request is validated and saved. If there is a validation error it to go back to the view page the request originated from. Visits are the same, validate then either make a new visit object or find an existing one then save the new data to the database. Once it has finished it redirects back to the index.

```php
<?php

public function store(Request $request)
    {
        $request->validate([
            'name' => 'required',
            'email' => 'required|email',
            'postal_address' => 'required|max:6|min:6',
            'phone_number' => 'required|regex:/[0-9]{11}/',
            'start_date' => 'required|date',
        ]);

        $doctor = new Doctor();//makes a new doctor
        $doctor->name = $request->input('name');
        $doctor->email = $request->input('email');
        $doctor->postal_address = $request->input('postal_address');
        $doctor->phone_number = $request->input('phone_number');
        $doctor->start_date = $request->input('start_date');

        $doctor->save();

        return redirect()->route('admin.doctors.index');
    }

public function update(Request $request, $id)
```

```
    {
        $request->validate([

            'name' => 'required',
            'email' => 'required|email',
            'postal_address' => 'required|max:6|min:6',
            'phone_number' => 'required|regex:/[0-9]{11}/',
            'start_date' => 'required|date',
        ]);

        $doctor = Doctor::findOrFail($id);//finds existing doctor

        $doctor->name = $request->input('name');
        $doctor->email = $request->input('email');
        $doctor->postal_address = $request->input('postal_address');
        $doctor->phone_number = $request->input('phone_number');
        $doctor->start_date = $request->input('start_date');

        $doctor->save();

        return redirect()->route('admin.doctors.index');
    }
```

There is an exception for patients as its fields are only required if a checkbox has been checked. I put in custom error messages to display for the user because the default error messages does not make sense to a normal user. I am also checking if medical insurance is checked. If checked save the other fields dependant on medical insurance. if not don't bother saving them. Editing a patient is the exact same, validation is the same and saving it is the same. Only difference is finding an existing patient.

```
<?php

public function store(Request $request)
    {
        $request->validate([
            'name' => 'required',
            'email' => 'required|email|unique:patients,email',
            'postal_address' => 'required|min:6|max:6',
            'phone_number' => 'required|regex:/[0-9]{11}/',
            'medical_insurance' => 'nullable|boolean',
            'company' => 'nullable|required_if:medical_insurance,1|integer',
            'policy_number' =>
'nullable|required_if:medical_insurance,1|min:6|max:6'
        ],[
            'medical_insurance.boolean' => 'Medical Insurance must be a valid
Value.',
            'policy_number.required_if' => 'Policy Number is required.',
            'company.required_if' => 'Company is required.',
            'company.integer' => 'Company must be a valid value.'
        ]);

        $patient = new Patient();// would be  $patient = Patient::findOrFail($id)
for edit
        $patient->name = $request->input('name');
        $patient->email = $request->input('email');
        $patient->postal_address = $request->input('postal_address');
        $patient->phone_number = $request->input('phone_number');

        //if medical insurance is checked
        if(!empty($request->input('medical_insurance'))){
            $patient->medical_insurance = $request->input('medical_insurance');
            $patient->company_id = $request->input('company');
            $patient->policy_number = $request->input('policy_number');
        }else{
            $patient->medical_insurance = 0;
        }

        $patient->save();

        return redirect()->route('admin.patients.index');
    }
```

Delete

For delete I make sure that integrity is preserved, If a patient or doctor is being deleted I delete all of their respective visits first then I delete the patient or doctor. In this example a patient.

```
<?php
public function destroy($id)
    {
        $patient = Patient::findOrFail($id);

        Visit::where('patient_id', $patient->id)->delete();
        $patient->delete();

        return redirect()->route('admin.patients.index');
    }
```

After the patient is deleted it routes back to the index.

Home Controllers

I wanted to make sure that each role type has its own home page but I didn't want to get rid of the `/home` route it's too convenient to use. So in the base *HomeController* index function I put in if statements to redirect the logged in user to their correct home page.

```php
<?php

public function index()
    {
        $user = Auth::user();

        if($user->role->name == 'admin'){
            return view('admin.home');
        }else if($user->role->name == 'user'){
            return view('user.home');
        }
    }
```

Middleware

Middleware is code that is executed before the route goes to where it's supposed to go. I wanted to make it so that a user cannot access admin routes and admins can't access user routes. First I needed to check if there is someone currently logged in as in Routes there is this `->middleware('auth')`. This comes already with laravel and only checks if someone is logged in. If not it brings them to the login url. I created two new pieces of middleware that checks if the person logged in is and admin or user.

In admin controllers the constructor looks like:

```php
<?php

public function __construct()
    {
        $this->middleware('checkAdmin');
    }
```

And users constructor looks like:

```php
<?php

public function __construct()
    {
        $this->middleware('checkUser');
    }
```

If a user tried to access an admin route, they will be shown an *Unauthorised Message*. In the middleware class for both admin and user:

```
<?php

public function handle($request, Closure $next)
    {
        //checks if there is someone logged in or if that someone has an admin role
or user role
        if (!$request->user() || !$request->user()->hasRole('admin')) {
            echo '<a href="/"><button>Back to Welcome Page</button></a><br><br>';
            return response('Unauthorised.', 401);
        }

        return $next($request);
    }
```

The check for if someone is logged in at all is overridden by the `auth` middleware, If I didn't use *auth*, a non-logged in user will see the message and not the log in page.

Now I have both middleware files set up, I need access to them. I edited the `Kernel.php` file accordingly:

```php <?php

protected $routeMiddleware = [

```
    /.../

    'checkAdmin' => \App\Http\Middleware\CheckAdmin::class,
    'checkUser' => \App\Http\Middleware\CheckUser::class,
];
```

```
<br>

### Views
View are the front end of the application. What the user gets to see and interact
with.

#### View Index and Show
Views use a templating language with syntax simlarities to Angular JS, views get
their data from the controller that returns them. For the index of doctors,
patients and visits. A foreach loop loops through each element in the array to
display all the respective objects but in patients I made it so that patients with
no medical insurance display information that makes sense.

```html

<td>
    @if($p->medical_insurance == 1)
        {{ 'Yes' }}
    @else
```

```
        {{ 'No' }}
    @endif
</td>

<td>
    @if($companies->where('id', $p->company_id)->first() == null)
        {{ 'N/A' }}
    @else
        {{$companies->where('id', $p->company_id)->first()->name }}
    @endif
</td>

<td>
    @if($p->policy_number == null)
        {{ 'N/A' }}
    @else
        {{ $p->policy_number }}
    @endif
</td>
```

I made it so that doctor and patient names are displayed in visits.

View Create and Edit

Both create and edit forms have a hidden input field with a CSRF(Cross Site Request Forgery) token. The data sent to and from the server maybe secure but the source is not. The token ensures that the request made does come from the same source.

```
<input type="hidden" name="_token" value="{{ csrf_token() }}">
```

For the create view it's mainly just a form with inputs matching those of the fields of the table in the database. When there are validation errors an *old* function is called which displays the values before the user submitted the form. For edit forms the old function is the same but takes a second parameter, the value of the object being edited. It displays if there is no old value.

Edit forms also require a PUT method. HTML don't support this so another hidden input field must be put in that laravel itself interprets as such:

```
<input type="hidden" name="_method" value="PUT">
```

View Delete

Delete routes do not have a view file but I added JQuery confirmation when deleting a doctor, patient or visit. In the *app.blade.php* at the end:

```
<script>


    $(document).ready(function(){

        //toDelete class on delete forms checks on submit
        $(".toDelete").one("submit", function(e){
            e.preventDefault();//prevents the form from sending

            var url = $(this)["0"].baseURI;//gets the url of the delete form

            //checks agains the url to dislpay the correct type of object to be
deleted
            if(url.indexOf("patients") > 0){
                item = "Patient";
            }else if(url.indexOf("doctors") > 0){
                item = "Doctor";
            }else if(url.indexOf("visits") > 0){
                item = "Visit";
            }

            proceed(item);

        });

        //function to proceed with the delete submit
        function proceed(type){

            if(confirm("Delete this " + type + "? This cannot be undone!")){
                $(".toDelete").off().submit();
            }
        }
    });

    </script>
```

Link to GitHub Repository Avaya Link