A Laravel prime: Part 3

A bit of maintenance first

So far we have worked with very little style applied to the page, and while this might be functional, you will want to know how to use some of the most common styling help frameworks available. For example, how do we use Bootstrap with our app?

You'll be glad to know that all you need to do is to include it in your master layout page. So you can edit main.blade.php in the layouts folder to include the bootstrap css stylesheet:

```
<link  rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css"
>
```

After this, you can simply start making use of the classes defined in the css. For example, if you have been using this code

```
<div class="container">
    @yield('content')
</div>
```

your content will automatically be set with some default margins. We can however improve the way our cards look in the detail section. For example, we can surround our code to present the card in divs like this:

```
<div class="row">
    <div class="col-md-6 col-md-offset-3">
        …
    </div>
</div>
```

so that the card and its notes will appear centred in the page. You can also apply a list group class to the unordered list tag, like this:

```
<ul class="list-group">
```

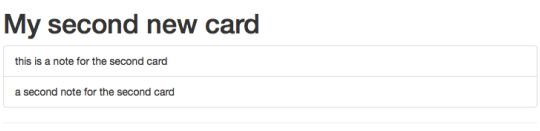
and then apply the list-group-item class to each list item:

```
<li class="list-group-item">
```

and you should see already quite a difference in the presentation of the card.

From this:

to this:



Some simple formatting but it makes it much more pleasant to work with. You may explore more about twitter bootstrap and its philosophy for organising content and making it "responsive" by looking in the bootstrap web page: http://getbootstrap.com

Adding notes to the cards

Ok, so let's say that while we are in the card view we want to be able to add notes to the card. How do we go about doing this with Laravel? Well, first of all, we are going to need a form. Let's build one one step at the time, while adding some formatting to it to make it look relatively nice. Add this code to your show.blade.php file, under the closing ul tag:

```
<hr>
<h3>Add a New note</h3>

<form>
    <div class="form-group">

        <textarea name="body" class="form-control"></textarea>

    </div>
    <div class="form-group">

        <button type="submit" class="btn btn-primary">Add note</button>

    </div>
</form>
```

If you reload the page, you should see a fairly nice form to add a new note to your card. However, if you try to use it at this point, nothing will happen. That's because when you are clicking the submit button, the form is by default sending a get request to the same page its originating from, and it's passing the body of your textarea in the request (you can check that this is the case by looking at the URL). We need to fix this.

For starters, we don't want to use a "GET" method. We want to use POST. Also, we don't want to POST to the same page we are at the moment. We need to decide how to handle this. You could decide to POST your form to any URI, for example, you may decide that this should be handled by a NotesController, so you would want to POST the form data to something like /notes, or you may want to handle that as part of the current card, so perhaps something like /cards/{card}/notes. However, you will want to be careful: either option is good, but beware of creating very complicated routes to handle your requests.

For our example, /cards/{card}/notes should be good enough. So, we need to tell the form where to send the request:

```
<form method="post" action="/cards/{{$card->id}}/notes">
```

But if you try this now, you will run into an error, because the route doesn't currently exist. So we need to fix that now. Let's add the route to the routes.php file:

```
Route::post('cards/{card}/notes','NotesController@store');
```

Again, here you have some options, you could send to a NotesController, or a CardNotesController, or really any other option you can think of that should descriptively work with your logic. For now, this should be good enough. Next we need to create this new NotesController, and add the store action. This should be familiar now:

```
php artisan make:controller NotesController
```

and then add the store method to the controller:

```
public function store () {

}
```

but how do we handle the request? Well, Laravel has created a "request" object for us, which will have the information being sent from the form. To prove we can actually read this object we can simply do this:

```
public function store ( Request $request ) {
    return $request->all();
}
```

Try to submit the form now, and see that you have access to this request object.

Great! So now we can proceed to create the note:

```
$note = new Note;
$note->body = $request->body;
```

But, how do we associate this with our card? Well, remember we also have access to the card, we have included the token in the route (the {card} part):

```
Route::post('cards/{card}/notes','NotesController@store');
```

So, we should be able to also get access to that in the store method. So we can then instantiate our new note, add the body, and save it to the card. When we are done, we simply want to send the user back to the card view and update with the new note:

```
public function store ( Request $request, Card $card ) {
    $note = new Note;
    $note->body = $request->body;
    $card->notes()->save($note);
    return back();
}
```

Try this and now you should have a working form. Notice we didn't have to do much to get this working. Just a few short lines of code. Here's another way to

solve the problem. Since we have access to the request and the card, we can actually **create** the new note directly, without having to instantiate one first:

```
public function store ( Request $request, Card $card ) {
    $card->notes()->create($request->all());
    return back();
}
```

However, if you try this, you are likely to run into a "Mass Assignment Exception" error. This is Laravel protecting you against potential code injection. Basically, Laravel is telling us that an attempt was made to mass assign values to a model where no such thing has been explicitly allowed. To fix this, we need to add some code to our model, to allow it. Modify your model for the Note like this:

```
class Note extends Model
{
    protected $fillable = ['body'];
    public function card(){
        return $this->belongsTo(Card::class);
    }
}
```

This $fillable property, tells Laravel that we are prepared to allow mass assignment, but ONLY on the 'body' field. Therefore, if an attempt is made to modify for example the note ID or the card ID, these will be ignored. Try the code once more, and this time you shouldn't have any problems.

There is still however one problem with our Form: try to add an empty note, and you should see that although it appears that nothing has been added, there is now an empty note in the database. We shouldn't be allowing this.

Validation

How do we protect ourselves against this sort of thing? Well, we just need to do a few simple modifications to allow Laravel to help us do a bit of validation before we add a note. First of all, we need to get access to the "$errors" message bag that Laravel provides. In order to do so, we need to move our routes into the group that uses the web middleware. We do that in the routes.php file: modify it like so:

```
Route::get('/', 'PagesController@home');

Route::get('about','PagesController@about');


/*…*/

Route::group(['middleware' => ['web']], function () {

    Route::get('cards', 'CardsController@index');

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

    Route::post('cards/{card}/notes','NotesController@store');
```

Notice that since we only need the message bag in those routes that generate dynamic content, we only move those, leaving our static pages out of the group.

Your application should currently work just as before. This isn't a major change, however, you should now have access to an $error object generated by Laravel. Let's see it. Add this code at the end of your form in the show.blade.php file:

```
</form>
{{ var_dump($errors) }}
```

When you reload the page you should see this under the form:

Great, half way there. So, how do we tell Laravel to do some validation for us? I'm going to show you the long way, but this is the way most large projects should do things. We will create a request object, and we will use it to add our validation there.

```
php artisan make:request AddNoteRequest
```

This will create a new type of request, our AddNoteRequest. This class has two methods, an authorize method, used to define who should have access to this object; and a rule method, which will contain our validation rules. We are currently not using authentication of any sort, we don't even have users yet, so we can simply change the return line in that method for

```
return true;
```

Our rule method should return just an array with our validation rules:

```
public function rules()
{
    return [
        'body' => 'required|min:5'
    ];
}
```

All we are saying here is that the body of the note is required, and just for fun, I have set a minimum size of 5 characters. To find out more about validation Rules in Laravel check the documentation:

https://laravel.com/docs/5.2/validation#available-validation-rules

You can also write the validation rules as a classic array:

```
public function rules()
{
    return [
        'body' => ['required','min:5']
    ];
}
```

either way is fine.

Now all we need to do is tell Laravel to use our new Request class instead of the default one. So in the Notes controller, change the store method so that it makes use of the new class:

```
public function store ( AddNotesRequest $request, Card $card ) {
```

What is important to remember is that once we type hint the class, Laravel will do its best to try to give us that object, triggering the validation in the way. If the Validation passes, the object gets created and is available to use in the store method. If the validation fails, then the store method never gets executed, Laravel returns control to the previous request, and flashes the error in the error bag. Which is what we will take advantage off now. In the show.blade.php file we need to handle the error now. For the moment we can just display any errors at the bottom of the form. Modify the code so it looks like this:

```
</form>

    @if($errors->any())
        <ul class="alert alert-danger">
            @foreach($errors->all() as $error)
                <li class="list-group-item">{{ $error }}</li>
            @endforeach
        </ul>
    @endif
</div>
```

There is likely one more error to fix at the moment. Laravel by default includes a token on every request. This token is intended to be use to avoid Cross Site Request Forgery or CSRF. We are not making use of this, so we need to tell Laravel to allow requests without the token. This is done by changing the class VerifyCsrfToken in the App\Http\Middleware folder:

```
protected $except = [
    'cards/*'
];
```

Or, you can make use of the token by adding the csrf token to the form, like this:

```
<form method="post" action="/cards/{{$card->id}}/notes">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
```

Restful Resources

We have already created several routes to different pages that will do different things. What we have been creating is called "a RESTful Controller" or in Laravel jargon a Resource.

Let's see some more about resources. In your terminal, run this command:

```
php artisan route:list
```

Compare the result with this image for a Photo Resource:

Verb	Path	Action	Route Name
GET	/photo	index	photo.index
GET	/photo/create	create	photo.create
POST	/photo	store	photo.store
GET	/photo/{photo}	show	photo.show
GET	/photo/{photo}/edit	edit	photo.edit
PUT/PATCH	/photo/{photo}	update	photo.update
DELETE	/photo/{photo}	destroy	photo.destroy

Figure 1 A Sample RESTful resource

Resources follow a specific pattern, which is considered a standard. The image shows all the appropriate "verbs" and their expected result. For example, if you attempted to GET the resource /photo you would expect to see the index of all the photos in your system (the action column). Therefore the route you should use to service such request will be named photo.index. A GET request to /photo/create should give you the form needed to create a new photo, serviced by the route photo.create, this form would then POST the form data to /photo to store a new photo in your system (the store action), and the route to service this request would be photo.store; etc.

Laravel excels at helping you code the way you want. There is a way you can specify that a controller follows this standard which saves you from explicitly defining every route by hand. You can try this instead, replace all the routes you have generated so far in this tutorial for the cards with this one:

```
Routes::resource('Cards, 'CardsController');
```

Save the file and in your terminal execute again

```
php artisan route:list
```

Compare this with the routes you had before. Notice any difference? You may need to correct a few little things to make things work perfectly, but in the whole, you have already been following this train of thought. You should notice that not only you still have the routes you previously defined, but you also have new routes already associated with methods in your controller (which you now need to explicitly implement).

Also

The Laravel documentation has samples of other relationships in the Eloquent model, which are listed below:

One To One -> hasOne() and belongsTo().
One To Many -> hasMany() and belongsTo().
Many to Many -> belongsToMany() in both directions.
Has Many Through -> hasManyThrough()
Polymorphic
Many to Many polymorphic

You are encouraged to read the documentation to understand as a minimum the One to One, One to Many and Many to Many relationships.