Laravel
Practical 2

James Ooi

Feb 2018

Contents

1 Some Hints for Practical 01 Homework

Before we begin the practical topic for Practical 02, here are some hints for the *homework* you should have completed before this class. Check your solution to see if you have implemented the operations for the **members** table is done in a proper way.

1.1 Using Radio Buttons for *gender* Attribute

You should be using *radio buttons* for the **gender** attribute. In our **members** table, the **gender** attribute is of type **CHAR(1)**, we will be storing a single character '**M**' for male and '**F**' for female.

You should define an array named **$genders** in the **App\Common** class just like what you have done for **$states** and use it to generate the radio buttons in your view templates.

In the view templates, you should be using a **foreach** loop to generate the radio buttons:

```
1 @foreach(Common::$genders as $key => $val)
2   {!! Form::radio('gender', $key) !!} {{$val}}
3 @endforeach
```

1.2 Using Dropdown List for *division_id* Attribute

Take note that the **division_id** attribute is a *foreign key* that references the **id** attribute of **divisions** table. For this attribute, we will be storing an unsigned integer. But where do we get the values from? We cannot define the values in the **App\Common** class because the data for the divisions are to be entered by a user.

Instead, our data will be retrieved from the **divisions** table. To do so, we use the **pluck** method of the Eloquent model to retrieve the required data as an associative array which we can then use to generate the dropdown list:

```
1 {!! Form::select('division_id',
2   Division::pluck('name', 'id'),
3   null, [
4     'class'         => 'form-control',
5     'placeholder'   => '- Select Division -',
6 ]) !!}
```

In the above code, take note that the **pluck** method takes two parameters. The first parameter specifies the attribute of the model which will be used as the *value*, in this case the **name** attribute. The second parameter specifies the *key*, in this case the **id** attribute. The returned associative array has the data sorted by the *value*.

2 Resource Controllers & Routing

2.1 Resource Controllers

A feature of Laravel controllers is the use of *resource routing* which assigns the typical "CRUD" routes to a controller with a single line of code.

Let's create a resource controller that handles all HTTP requests for **groups** stored by our application with the following Artisan command:

```
php artisan make:controller GroupController --resource
```

This will generate the **GroupController** as follows:

```
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6
7  class GroupController extends Controller
8  {
9      /**
10      * Display a listing of the resource.
11      *
12      * @return \Illuminate\Http\Response
13      */
14     public function index()
15     {
16         //
17     }
18
19     /**
20      * Show the form for creating a new resource.
21      *
22      * @return \Illuminate\Http\Response
23      */
24     public function create()
25     {
26         //
27     }
28
29     /**
30      * Store a newly created resource in storage.
31      *
32      * @param  \Illuminate\Http\Request  $request
33      * @return \Illuminate\Http\Response
34      */
35     public function store(Request $request)
```

```
36      {
37          //
38      }
39
40      /**
41       * Display the specified resource.
42       *
43       * @param  int  $id
44       * @return \Illuminate\Http\Response
45       */
46      public function show($id)
47      {
48          //
49      }
50
51      /**
52       * Show the form for editing the specified resource.
53       *
54       * @param  int  $id
55       * @return \Illuminate\Http\Response
56       */
57      public function edit($id)
58      {
59          //
60      }
61
62      /**
63       * Update the specified resource in storage.
64       *
65       * @param  \Illuminate\Http\Request  $request
66       * @param  int  $id
67       * @return \Illuminate\Http\Response
68       */
69      public function update(Request $request, $id)
70      {
71          //
72      }
73
74      /**
75       * Remove the specified resource from storage.
76       *
77       * @param  int  $id
78       * @return \Illuminate\Http\Response
79       */
80      public function destroy($id)
81      {
82          //
83      }
84  }
```

You will notice that the following seven action methods are defined:

- **index**
- **show**
- **create**
- **store**
- **edit**
- **update**
- **destroy**

In the previous practical, we have manually defined the abovementioned methods in our **DivisionController** and **MemberController** controller classes, except the the **destroy** method.

By generating a resource controller, it allows developers to use standard method names for controller actions to implement CRUD operations, thereby ensuring consistency.

2.2 Resource Routing

Irrespective whether you have generated the resource controller using Artisan or have manually coded them, as long as the methods names implementing CRUD operations follow the above convention, you can define a **resource route** to specify routes to all CRUD operations in a single line:

```
1 Route::resource('/group', 'GroupController');
```

With the above routing defined, the following routes will be created:

No.	Verb	URI	Action Method	Route Name
1	GET	/group	index	group.index
2	GET	/group/create	create	group.create
3	POST	/group	store	group.store
4	GET	/group/id	show	group.show
5	GET	/group/id/edit	edit	group.edit
6	PUT	/group/id	update	group.update
7	DELETE	/group/id	destroy	group.destroy

Take note that the route names are also automatically made available to us, which is the same convention we have used when defining the routes for the methods in **DivisionController** and **MemberController**.

You can also define resource routing for multiple resource controller as follows:

```
Route::resources([
    '/division' => 'DivisionController',
    '/member' => 'MemberController',
    '/group' => 'GroupController',
]);
```

In our scenario, we will not be implementing the **destroy** method in all three controllers. Hence, we will need to exclude them in our route definition. In **routes/web.php**, define the resource routes as follows:

```
Route::resource('/division', 'DivisionController', ['except' => [
    'destroy',
]]);

Route::resource('/member', 'MemberController', ['except' => [
    'destroy',
]]);

Route::resource('/group', 'GroupController', ['except' => [
    'destroy',
]]);
```

Take note that in the third parameter, we specify the methods that will be excluded from the list of routes generated. Laravel also allows us to specify methods that are included using the **only** key.

Comment or remove the routes that you have previously defined for all actions of **DivisionController** and **MemberController**.

2.3 Difference between *store* and *update*

Using resource routes, Laravel defines different HTTP verbs for the **store** and **update** methods. While the **store** method uses the HTTP **POST** verb, the **update** method uses the HTTP **PUT** verb.

If you try to access your app for both the **DivisionController** and **MemberController** actions, you will notice that while everything should work, the updating of rows failed!

As the the **update** method uses the HTTP **PUT** verb, you will need to specify that the form should use the **PUT** verb.

Using Laravel Collective Forms & HTML:

```
{!! Form::model($division, [
        'route' => ['division.update', $division->id],
        'method' => 'put',
        'class' => 'form-horizontal'
]) !!}
```

In case you are not using Laravel Collective but just plain HTML:

```
{{ method_field('PUT') }}
```

The above code shows what you would specify in the **edit.blade.php** view template for **DivisionController**.

This will result in a hidden HTML input field being included in the form as follows:

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

You are also required to make the similar modifications to the **edit.blade.php** view template for **MemberController**.

Once you have completed the necessary changes, test your application again to ensure all the functionalities work as intended.

3 More on Models & Relationships

3.1 Prepping the Database for groups

Once again, we will begin using *migrations* to create the following tables:

3.1.1 The *groups* Table

We shall now build a table to hold the *groups* of the club. Create the following migration:

```
php artisan make:migration create_groups_table --create=groups
```

Write the following code for the abovementioned migration **up** method:

```
1  public function up()
2  {
3      Schema::create('groups', function (Blueprint $table) {
4          $table->engine = 'InnoDB';
5          $table->increments('id');
6          $table->char('code', 3)->unique();
7          $table->string('name', 100)->index();
8          $table->text('description')->nullable();
9          $table->timestamps();
10     });
11 }
```

3.1.2 The Pivot Table

In our scenario, a member can join multiple groups and a group may have more than one member, hence, a *many-to-many* relationship. We will need a pivot table between the **groups** and **members** table to implement the *many-to-many* relationship.

In Laravel, the name of the pivot table is derived from the alphabetical order of the related model names. Hence, we will need to create a table named **group_member**. The pivot table consists of at least two columns, **member_id** and **group_id**, which references the corresponding primary key in the two related tables.

```
php artisan make:migration create_group_member_table --create=group_member
```

```
1  public function up()
2  {
3      Schema::create('group_member', function (Blueprint $table) {
4          $table->unsignedInteger('member_id');
5          $table->unsignedInteger('group_id');
6
7          $table->foreign('member_id')
```

```
8                   ->references('id')->on('members');
9           $table->foreign('group_id')
10                  ->references('id')->on('groups');
11      });
12  }
```

3.1.3 Relationships

Next, using Artisan CLI, generate a model named **Group** for the **groups** table. Take note that you do NOT need to generate a model for the pivot table!

Now, we shall define the required relationships between the **Member** and **Group** models.

In our **Member** model class, define the following relationship method:

```
1  /**
2   * The groups that belong to the member.
3   */
4  public function groups()
5  {
6      return $this->belongsToMany('Group');
7  }
```

In our **Group** model class, define the following relationship method:

```
1  /**
2   * The members that belong to the group.
3   */
4  public function members()
5  {
6      return $this->belongsToMany('Member');
7  }
```

3.1.4 Custom Name for Pivot Table

What if the name of your pivot table does not correspond to Laravel's convention? Say, if your pivot table for the abovementioned scenario is named **member_group_rel** instead. You can override the default naming convention by specifying the table name as the second parameter in the **belongsToMany** method:

```
1  /**
2   * The members that belong to the group.
3   */
4  public function members()
5  {
6      return $this->belongsToMany('Member', 'member_group_rel');
7  }
```

3.2 Custom Table Name

As an additional note, you can also override the default table naming convention for your models by specifying the **table** attribute in your model class:

```
1 class Group extends Model
2 {
3   protected $table = 'group';
4 }
```

4 Practical Tasks (Unguided)

- Complete the required create, retrieve and update operations for the **groups** table. You are required to define the necessary *mass-assignable* attributes for the **Group** model in completing this task.
- Create the functionality to assign one or more groups to a member in the **MemberController**.
- Create the functionality to add one or more members to a group in the **GroupController**.