Laravel

Practical 5

James Ooi

Mar 2018

Contents

1 Authentication

In this section, we shall begin implementing authentication for our Travelife Club application. Before proceeding, ensure that your Laravel project already contains the **User** model class defined and the table **users** is already exist in the database.

1.1 Execute the Artisan *make:auth* Command

Execute the following Artisan CLI command:

```
php artisan make:auth
```

This will generate the required controller files, views and routes that are required for authentication.

1.2 Authentication Routes

You should notice that the routes for authentication are defined in your route file with the following statement:

```
1 Auth::routes();
```

This is similar to defining the following routes individually:

```
1  Route::get('login', 'Auth\LoginController@showLoginForm')
2          ->name('login');
3  Route::post('login', 'Auth\LoginController@login');
4  Route::post('logout', 'Auth\LoginController@logout')
5          ->name('logout');
6
7  // Registration Routes...
8  Route::get('register', 'Auth\RegisterController@showRegistrationForm')
9          ->name('register');
10 Route::post('register', 'Auth\RegisterController@register');
11
12 // Password Reset Routes...
13 Route::get('password/reset', 'Auth\ForgotPasswordController@showLinkRequestForm')
14         ->name('password.request');
15 Route::post('password/email', 'Auth\ForgotPasswordController@sendResetLinkEmail')
16         ->name('password.email');
17 Route::get('password/reset/{token}', 'Auth\ResetPasswordController@showResetForm')
18         ->name('password.reset');
19 Route::post('password/reset', 'Auth\ResetPasswordController@reset');
```

You will notice from the above that the routes generated included routes for logging in and

logging out, user registration and password resetting features. These features are essential when you are developing a web app that will be accessible by the public, such as a social media app.

However, when developing a web-based app that are for internal organizational use, no user registration is needed. In fact, there should NOT be any feature for user registration as users are created by the system administrator. In such a case, you will need to build the necessary CRUD operations to maintain users data, which will be used by an existing user with the necessary access rights.

In many cases, for organizational use, password resetting feature may not be required too, subject to organizational policies. However, this feature is very essential on web apps that are used by the public.

As our Travelife Club app is meant for internal organizational use, we only need to implement the logging in and logging out feature, as well as the CRUD operations to maintain users data.

Remove the `Auth::routes();` statement in your route file and enter the following statements instead:

```
1 Route::get('login', 'Auth\LoginController@showLoginForm')->name('login');
2 Route::post('login', 'Auth\LoginController@login');
3 Route::post('logout', 'Auth\LoginController@logout')->name('logout');
```

This will define only routes for logging in and logging out feature, including to show the login form.

Not shown in the above code are the routes for the CRUD operations of users, which you will be working on unguided as your homework!

1.3 Creating the First User Account

We shall use Laravel's migration to create the first user account for our app. Write the following code in the **up** method of your migration file and execute your migration to create the initial user:

```
1  /**
2   * Run the migrations.
3   *
4   * @return void
5   */
6  public function up()
7  {
8      DB::table('users')->insert([
9          'email' => 'admin@admin.com',
10         'password' => bcrypt('Adm1nP@$$'),
11         'name' => 'Administrator',
12     ]);
13 }
```

Go to the URL **http://localhost:8088/login** and try logging in using the user account created above.

NOTE: If your app is for public access and uses the user registration and password resets feature, the code generated using `make:auth` Artisan CLI command should also work without any coding effort!

1.4 Protecting Routes

Laravel ships with an **auth** middleware, which is defined at **Illuminate\Auth\Middleware\Authenticate**. Since this middleware is already registered in your HTTP kernel, all you need to do is attach the middleware to a route definition:

```
1 Route::get('profile', function () {
2     // Only authenticated users may enter...
3 })->middleware('auth');
```

If you are using controllers, you may call the **middleware** method from the controller's constructor instead of attaching it in the route definition directly:

```
1 public function __construct()
2 {
3     $this->middleware('auth');
4 }
```

We shall protect all actions in the three controllers you have created in earlier sessions: **DivisionController**, **MemberController** and **GroupController**.

Define the **constructor method** in all the abovementioned controllers as follows:

```
1 public function __construct()
2 {
3     $this->middleware('auth');
4 }
```

Once you have done the above, try accessing the actions for the abovementioned controllers. You will notice that you can only access them once user is logged in.

2 Authorization

You have implemented authentication with very little effort! However, in many applications, authentication itself is insufficient. In a blogging platform, for example, only the author himself is authorized to perform update and delete operations on his own blog posts. If we simply allow any authenticated user to access, then all authenticated users can update and delete any blog posts!

In our Travelife Club app, we would be having three types of users:

- **staff**.
- **manager**.
- **admin**.

We would implement authorization based on the following rules:

- Only **admin** and **manager** users can create, update and delete divisions and groups.
- All users may retrieve/view divisions and groups.
- All users may create, update, delete and retrieve members.

2.1 Roles and Abilities

Laravel provides us with a simple way to authorize user actions against a given resource. Like authentication, Laravel's approach to authorization is simple, and there are two primary ways of authorizing actions: **gates** and **policies**, which will define the authorization rules that we required.

Laravel does not provide us with the feature to manage the **roles** and **abilities**. You can of course create the necessary tables and CRUD operations to manage the roles and abilities for authorization purpose. However, there are various external packages that can assist us in implementing roles and abilities easily!

2.1.1 Bouncer

One of the packages for implemeting roles and abilities is **Bouncer**. Bouncer is an elegant, framework-agnostic approach to managing roles and abilities for any app using Eloquent models. With an expressive and fluent syntax, it stays out of your way as much as possible: use it when you want, ignore it when you don't.

More information about Bouncer is available at https://github.com/JosephSilber/bouncer.

2.1.1.1 Installating Bouncer

Install Bouncer with composer:

```
composer require silber/bouncer v1.0.0-rc.1
```

2.1.1.2 Add the trait to User class

Once installation is completed, add the **HasRolesAndAbilities** trait to your **User** model class:

```
1 use Silber\Bouncer\Database\HasRolesAndAbilities;
2
3 class User extends Model
4 {
5         use HasRolesAndAbilities;
6
7         //.....
8 }
```

2.1.1.3 Running Bouncer Migration

Run the migration to create the required tables for Bouncer:

```
php artisan vendor:publish --tag="bouncer.migrations"
```

```
php artisan migrate
```

This will create the following tables in your database:

- **abilities**.
- **roles**.
- **permissions**.
- **assigned_roles**.

2.1.1.4 Creating Roles and Abilities

Create a migration to initialize the roles, abilities and permissions:

```
php artisan make:migration init_roles_and_permissions
```

Write the following code in your migration file:

```php
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
use App\User;

class InitRolesAndPermissions extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        // Define roles
        $admin = Bouncer::role()->create([
            'name' => 'admin',
            'title' => 'Administrator',
        ]);

        $manager = Bouncer::role()->create([
            'name' => 'manager',
            'title' => 'Manager',
        ]);

        $staff = Bouncer::role()->create([
            'name' => 'staff',
            'title' => 'Staff',
        ]);

        // Define abilities
        $viewMember = Bouncer::ability()->create([
            'name' => 'view-member',
            'title' => 'View Member',
        ]);

        $createMember = Bouncer::ability()->create([
            'name' => 'create-member',
            'title' => 'Create Member',
        ]);

        $manageMember = Bouncer::ability()->create([
            'name' => 'manage-member',
            'title' => 'Manage Member',
        ]);

        $viewDivision = Bouncer::ability()->create([
            'name' => 'view-division',
```

```
            'title' => 'View Division',
        ]);

        $createDivision = Bouncer::ability()->create([
            'name' => 'create-division',
            'title' => 'Create Division',
        ]);

        $manageDivision = Bouncer::ability()->create([
            'name' => 'manage-division',
            'title' => 'Manage Division',
        ]);

        $viewGroup = Bouncer::ability()->create([
            'name' => 'view-group',
            'title' => 'View Group',
        ]);

        $createGroup = Bouncer::ability()->create([
            'name' => 'create-group',
            'title' => 'Create Group',
        ]);

        $manageGroup = Bouncer::ability()->create([
            'name' => 'manage-group',
            'title' => 'Manage Group',
        ]);

        // Assign abilities to roles
        Bouncer::allow($staff)->to($viewMember);
        Bouncer::allow($staff)->to($createMember);
        Bouncer::allow($staff)->to($manageMember);
        Bouncer::allow($staff)->to($viewDivision);
        Bouncer::allow($staff)->to($viewGroup);

        Bouncer::allow($manager)->to($viewMember);
        Bouncer::allow($manager)->to($createMember);
        Bouncer::allow($manager)->to($manageMember);
        Bouncer::allow($manager)->to($viewDivision);
        Bouncer::allow($manager)->to($createDivision);
        Bouncer::allow($manager)->to($manageDivision);
        Bouncer::allow($manager)->to($viewGroup);
        Bouncer::allow($manager)->to($createGroup);
        Bouncer::allow($manager)->to($manageGroup);

        Bouncer::allow($admin)->to($viewMember);
        Bouncer::allow($admin)->to($createMember);
        Bouncer::allow($admin)->to($manageMember);
        Bouncer::allow($admin)->to($viewDivision);
        Bouncer::allow($admin)->to($createDivision);
```

```
        Bouncer::allow($admin)->to($manageDivision);
        Bouncer::allow($admin)->to($viewGroup);
        Bouncer::allow($admin)->to($createGroup);
        Bouncer::allow($admin)->to($manageGroup);

        // Make the first user an admin
        $user = User::find(1);
        Bouncer::assign('admin')->to($user);
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        //
    }
}
```

Now, run the migration.

2.2 Policies

Policies are classes that organize authorization logic around a particular model or resource. For example, in our Travelife Club application, we have a **Member** model and a corresponding **MemberPolicy** to authorize user actions such as creating or updating member records.

2.2.1 Creating Policies

2.2.1.1 Member Policy

Generate a policy named **MemberPolicy**:

```
php artisan make:policy MemberPolicy
```

The generated policy will be placed in the **app/Policies** directory.

Now, build your policy class with the code below:

```
<?php

namespace App\Policies;

use App\User;
use App\Member;
```

```
 7 use Bouncer;
 8 use Illuminate\Auth\Access\HandlesAuthorization;
 9
10 class MemberPolicy
11 {
12     use HandlesAuthorization;
13
14     /**
15      * Create a new policy instance.
16      *
17      * @return void
18      */
19     public function __construct()
20     {
21         //
22     }
23
24     /**
25      * Determine if the given user can view members.
26      *
27      * @param  \App\User  $user
28      * @return bool
29      */
30     public function view(User $user)
31     {
32         return $user->can('view-member');
33     }
34
35     /**
36      * Determine if the given user can create members.
37      *
38      * @param  \App\User  $user
39      * @return bool
40      */
41     public function create(User $user)
42     {
43         return $user->can('create-member');
44     }
45
46     /**
47      * Determine if the given user can manage members.
48      *
49      * @param  \App\User  $user
50      * @param  \App\Member $member
51      * @return bool
52      */
53     public function manage(User $user, Member $member)
54     {
55         return $user->can('manage-member');
56     }
```

```
57 }
```

2.2.1.2 Division Policy

Create a policy named **DivisionPolicy** as follows:

```
1  <?php
2
3  namespace App\Policies;
4
5  use App\User;
6  use App\Division;
7  use Bouncer;
8  use Illuminate\Auth\Access\HandlesAuthorization;
9
10 class DivisionPolicy
11 {
12     use HandlesAuthorization;
13
14     /**
15      * Create a new policy instance.
16      *
17      * @return void
18      */
19     public function __construct()
20     {
21         //
22     }
23
24     /**
25      * Determine if the given user can view divisions.
26      *
27      * @param  \App\User  $user
28      * @return bool
29      */
30     public function view(User $user)
31     {
32         return $user->can('view-division');
33     }
34
35     /**
36      * Determine if the given user can create divisions.
37      *
38      * @param  \App\User  $user
39      * @return bool
40      */
41     public function create(User $user)
42     {
43         return $user->can('create-division');
```

```
44      }
45
46      /**
47       * Determine if the given user can manage divisions.
48       *
49       * @param  \App\User  $user
50       * @param  \App\Division $division
51       * @return bool
52       */
53      public function manage(User $user, Division $division)
54      {
55          return $user->can('manage-division');
56      }
57  }
```

2.2.1.3 Group Policy

Now, create the **GroupPolicy**:

```
1   <?php
2
3   namespace App\Policies;
4
5   use App\User;
6   use App\Group;
7   use Bouncer;
8   use Illuminate\Auth\Access\HandlesAuthorization;
9
10  class GroupPolicy
11  {
12      use HandlesAuthorization;
13
14      /**
15       * Create a new policy instance.
16       *
17       * @return void
18       */
19      public function __construct()
20      {
21          //
22      }
23
24      /**
25       * Determine if the given user can view groups.
26       *
27       * @param  \App\User  $user
28       * @return bool
29       */
30      public function view(User $user)
```

```
31      {
32          return $user->can('view-group');
33      }
34
35      /**
36       * Determine if the given user can create groups.
37       *
38       * @param  \App\User  $user
39       * @return bool
40       */
41      public function create(User $user)
42      {
43          return $user->can('create-group');
44      }
45
46      /**
47       * Determine if the given user can manage groups.
48       *
49       * @param  \App\User  $user
50       * @param  \App\Group $group
51       * @return bool
52       */
53      public function manage(User $user, Group $group)
54      {
55          return $user->can('manage-group');
56      }
57  }
```

Next, register your policies in the **AuthServiceProvider** class file:

```
1  /**
2   * The policy mappings for the application.
3   *
4   * @var array
5   */
6  protected $policies = [
7      Member::class => MemberPolicy::class,
8      Division::class => DivisionPolicy::class,
9      Group::class => GroupPolicy::class,
10 ];
```

2.2.2 Authorizing Actions

Once we have completed the above, we can now use our policies to authorize a user to perform an action in our controllers. To authorize an action we call the **authorize** method in the appropriate methods in our controllers.

To authorize a user to access the 'index' method in 'MemberController', do the following:

```
1 public function index(Request $request)
2 {
3     $this->authorize('view', Member::class);
4
5     // The rest of your code goes here...
6 }
```

In the above code, we specify **view** in the first parameter and the class name of **Member** Eloquent class as the second parameter. This means, we will be using the **view** method if the **MemberPolicy** to authorize actions.

The above example uses actions that do not require models. For actions that require models, we will pass a **Member** model instance as the second parameter. An example will be to authorize a user to access the 'edit' method in 'MemberController':

```
1 public function edit($id)
2 {
3     $member = Member::find($id);
4     if(!$member) throw new ModelNotFoundException;
5
6     $this->authorize('manage', $member);
7
8     // The rest of your code goes here...
9 }
```

2.2.3 Practical Exercise

Now, apply the appropriate methods of your policies to all the actions in **MemberController**, **DivisionController** and **GroupController**.

3 Exercise (Unguided)

Create the CRUD operations to create, view and edit users. You are required to use **Bouncer** to define the required *abilities* and *permissions* for these operations where only an **admin** user is allowed to access all these operations.

In your create and edit forms, allow the admin user to also assign a role to a user. In our app, one user will be assigned only one role. Hence, you may use a dropdown list to allow an admin to select the role to assign to the user.

When storing and updating, there are two ways you may use to store/update the role. First, you may define a **Role** Eloquent class and the appropriate relational methods in **Role** and **User** using the **assigned_roles** table as the pivot table and use the relational inserts/updates to insert/update the role.

Alternatively, you may use **Bouncer** to achieve the above.