User Management in Laravel

Christian Anthony Setyawan *13514085*
Teknik Informatika, Institut Teknologi Bandung
canthonysetyawan@gmail.com

***Abstract*—User management in Laravel is good, but not great. Laravel provides Authentication module to manage user login and authentication. The problem is that Authentication module doesn't support role model in users, eg. admin, normal user, user with more authority than normal user, etc. Entrust solves that(https://github.com/Zizaco/entrust). Entrust is a succinct and flexible way to add Role-based Permissions to Laravel.**

***Index Terms*—Laravel, PHP, user management, role-based user, additional package.**

I. Introduction

A web application framework is a software framework that is designed to support the development of dynamic websites, web applications and web services. The framework aims to alleviate the overhead associated with common activities performed in Web development.

Laravel is a web application framework with expressive, elegant syntax. Laravel attempts to take the pain out of development by easing common tasks used in the majority of web projects, such as authentication, routing, sessions, and caching.

Laravel makes implementing authentication very simple. In fact, almost everything is configured out of the box. At its core, Laravel's authentication facilities are made up of "guards" and "providers". Guards define how users are authenticated for each request. For example, Laravel ships with a session guard which maintains state using session storage and cookies.

Unfortunately, Laravel doesn't ship with role and permission management. Entrust solves that. Entrust is a succinct and flexible way to add Role-based Permissions to Laravel 5. This package provides a flexible way to add Role-based Permissions to Laravel

II. Entrust Package

A. Installation

Before installing Entrust package, Laravel and PHP must be installed first. Then after creating a new app in Laravel, Entrust package can be installed. To install Entrust package:

1) Add the following to *composer.json*:
   `"zizaco/entrust": "5.2.x-dev"`
   Then run
   `$ composer update`
2) Open `config/app.php` and add the following to the `providers` array:
   `Zizaco\Entrust\EntrustServiceProvider::class,`
3) In the same `config/app.php` and add the following to the `aliases` array:
   ```
   'Entrust'   =>
   ↪  Zizaco\Entrust\EntrustFacade::class,
   ```
4) Run the command below to publish the package config file `config/entrust.php`:
   `$ php artisan vendor:publish`
5) Open `config/auth.php` and add the following to it:
   ```
   'providers' => [
       'users' => [
           'driver' => 'eloquent',
           'model' =>
   ↪  Namespace\Of\Your\User\Model\User::class,
           'table' => 'users',
       ],
   ],
   ```

III. Configuration

Set the property values in the `config/auth.php`.
These values will be used by entrust to refer to the correct user table and model.

To further customize table names and model namespaces, edit the `config/entrust.php`.

1) User relation to roles: Now generate the Entrust migration:

```
php artisan entrust:migration
```

It will generate the `<timestamp>_entrust_setup_tables.php` migration.
You may now run it with the artisan migrate command:

```
php artisan migrate
```

After the migration, four new tables will be present:

- `roles` — stores role records
- `permissions` — stores permission records
- `role_user` — stores many-to-many relations between roles and users
- `permission_role` — stores many-to-many relations between roles and permissions

a) Role:

Create a Role model inside `app/models/Role.php` using the following example:

```
<?php namespace App;

use Zizaco\Entrust\EntrustRole;

class Role extends EntrustRole
{
}
```

The `Role` model has three main attributes:

- `name` — Unique name for the Role, used for looking up role information in the application layer. For example: "admin", "owner", "employee".
- `display_name` — Human readable name for the Role. Not necessarily unique and optional. For example: "User Administrator", "Project Owner", "Widget Co. Employee".
- `description` — A more detailed explanation of what the Role does. Also optional.

Both `display_name` and `description` are optional; their fields are nullable in the database.

b) Permission:

Create a Permission model inside `app/models/Permission.php` using the following example:

```
<?php namespace App;

use Zizaco\Entrust\EntrustPermission;

class Permission extends EntrustPermission
{
}
```

The `Permission` model has the same three attributes as the `Role`:

- `name` — Unique name for the permission, used for looking up permission information in the application layer. For example: "create-post", "edit-user", "post-payment", "mailing-list-subscribe".
- `display_name` — Human readable name for the permission. Not necessarily unique and optional. For example "Create Posts", "Edit Users", "Post Payments", "Subscribe to mailing list".
- `description` — A more detailed explanation of the Permission.

In general, it may be helpful to think of the last two attributes in the form of a sentence: "The permission `display_name` allows a user to `description`."

c) User:

Next, use the `EntrustUserTrait` trait in your existing `User` model. For example:

```
<?php

use Zizaco\Entrust\Traits\EntrustUserTrait;

class User extends Eloquent
{
    use EntrustUserTrait;

    ...
}
```

This will enable the relation with `Role` and add the following methods `roles()`, `hasRole($name)`, `can($permission)`, and `ability($roles, $permissions, $options)` within your `User` model.

Don't forget to dump composer autoload

```
composer dump-autoload
```

IV. Usage

Let's start by creating the following `Role`s and `Permission`s:

```
$owner = new Role();
$owner->name         = 'owner';
$owner->display_name = 'Project Owner'; // optional
$owner->description  =
'User is the owner of a given project'; // optional
$owner->save();

$admin = new Role();
$admin->name         = 'admin';
$admin->display_name =
'User Administrator'; // optional
$admin->description  =
'User is allowed to manage and edit other users';
// optional
$admin->save();
```

Next, with both roles created let's assign them to the users.
Thanks to the `HasRole` trait this is as easy as:

```
$user = User::where('username', '=', 'michele')
->first();

// role attach alias
$user->attachRole($admin);
// parameter can be an Role object, array, or id

// or eloquent's original technique
$user->roles()->attach($admin->id); // id only
```

Now we just need to add permissions to those Roles:

```
$createPost = new Permission();
$createPost->name         = 'create-post';
$createPost->display_name = 'Create Posts';
// optional
```

```
// Allow a user to...
$createPost->description  =
'create new blog posts'; // optional
$createPost->save();

$editUser = new Permission();
$editUser->name         = 'edit-user';
$editUser->display_name =
'Edit Users'; // optional
// Allow a user to...
$editUser->description  = 'edit existing users';
// optional
$editUser->save();

$admin->attachPermission($createPost);

$owner->attachPermissions(array
($createPost, $editUser));
```

a) Checking for Roles & Permissions:

Now we can check for roles and permissions simply by doing:

```
$user->hasRole('owner');   // false
$user->hasRole('admin');   // true
$user->can('edit-user');   // false
$user->can('create-post'); // true
```

Both `hasRole()` and `can()` can receive an array of roles & permissions to check:

```
$user->hasRole(['owner', 'admin']);       // true
$user->can(['edit-user', 'create-post']); // true
```

By default, if any of the roles or permissions are present for a user then the method will return true.
Passing `true` as a second parameter instructs the method to require **all** of the items:

```
$user->hasRole(['owner', 'admin']);
// true
$user->hasRole(['owner', 'admin'], true);
// false, user does not have admin role
$user->can(['edit-user', 'create-post']);
// true
$user->can(['edit-user', 'create-post'], true);
// false, user does not have edit-user permission
```

You can have as many `Roles` as you want for each `User` and vice versa.

The `Entrust` class has shortcuts to both `can()` and `hasRole()` for the currently logged in user:

```
Entrust::hasRole('role-name');
Entrust::can('permission-name');

// is identical to

Auth::user()->hasRole('role-name');
Auth::user()->can('permission-name');
```

You can also use placeholders (wildcards) to check any matching permission by doing:

```
// match any admin permission
$user->can("admin.*"); // true

// match any permission about users
$user->can("*_users"); // true
```

b) User ability:

More advanced checking can be done using the awesome `ability` function.
It takes in three parameters (roles, permissions, options):

- `roles` is a set of roles to check.
- `permissions` is a set of permissions to check.

Either of the roles or permissions variable can be a comma separated string or array:

```
$user->ability(array('admin', 'owner'),
array('create-post', 'edit-user'));

// or

$user->ability('admin,owner', 'create-post,edit-user');
```

This will check whether the user has any of the provided roles and permissions.
In this case it will return true since the user is an `admin` and has the `create-post` permission.

The third parameter is an options array:

```
$options = array(
    'validate_all' => true | false (Default: false),
    'return_type'  => boolean | array | both (
    Default: boolean)
);
```

- `validate_all` is a boolean flag to set whether to check all the values for true, or to return true if at least one role or permission is matched.
- `return_type` specifies whether to return a boolean, array of checked values, or both in an array.

Here is an example output:

```
$options = array(
    'validate_all' => true,
    'return_type' => 'both'
);

list($validate, $allValidations) = $user->ability(
    array('admin', 'owner'),
    array('create-post', 'edit-user'),
    $options
);

var_dump($validate);
// bool(false)
```

```php
var_dump($allValidations);
// array(4) {
//     ['role'] => bool(true)
//     ['role_2'] => bool(false)
//     ['create-post'] => bool(true)
//     ['edit-user'] => bool(false)
// }
```

The `Entrust` class has a shortcut to `ability()` for the currently logged in user:

```php
Entrust::ability('admin,owner', 'create-post,edit-user');

// is identical to

Auth::user()->ability('admin,owner', 'create-post,edit-user');
```

V. Conclusion

Tough Laravel already provides with a good Authentication module, Entrust leverage it with a great Role-based user management support.

Acknowledgment

The authors would like to thank God, Mr. Rinaldi, my parent, and my friends that are supporting me to make this paper.

References

[1] Entrust Package *https://github.com/Zizaco/entrust*

[2] Laravel *https://laravel.com*

Laravel Authentication *https://laravel.com/docs/5.4/authentication*