Laratrust Docs Documentation

Release 3.2.0

Santiago Garcia

Jun 27, 2017

Contents

Laratrust

CHAPTER 1

Table of Contents:

Note: Please read all the sections in order

Upgrade from 3.1 to 3.2

In order to upgrade from Laratrust 3.1 to 3.2 you have to follow these steps:

1. Change your `composer.json` to require the 3.2 version of laratrust:

   ```
   "santigarcor/laratrust": "3.2.*"
   ```

2. Run `composer update` to update the source code.
3. Add in your `config/laratrust.php` file this block:

   ```
   'user_models' => [
       'users' => 'App\User',
   ],
   ```

   And configure it with you user models information according to the new *Multiple User Models* explanation.
4. Run `php artisan laratrust:add-trait` to add the `LaratrustUserTrait` to the user models.
5. Run `php artisan laratrust:upgrade` to create the migration with the database upgrade.
6. Run `php artisan migrate` to apply the migration created in the previous step.
7. Delete the `LaratrustSeeder.php` file and run `php artisan laratrust:seeder`.
8. Run `composer dump-autoload`.

Now you can use the 3.2 version without any problem.

Installation

1. In order to install Laratrust in your Laravel project, just run the *composer require* command from your terminal:

   ```
   composer require "santigarcor/laratrust:3.2.*"
   ```

2. Then in your `config/app.php` add the following to the providers array:

   ```
   Laratrust\LaratrustServiceProvider::class,
   ```

3. In the same `config/app.php` add the following to the `aliases` array:

   ```
   'Laratrust'   => Laratrust\LaratrustFacade::class,
   ```

4. Run the next command to publish all the configuration files:

   ```
   php artisan vendor:publish --tag="laratrust"
   ```

5. If you are going to use *Middleware* (requires Laravel 5.1 or later) you also need to add the following to `routeMiddleware` array in `app/Http/Kernel.php`:

   ```
   'role' => \Laratrust\Middleware\LaratrustRole::class,
   'permission' => \Laratrust\Middleware\LaratrustPermission::class,
   'ability' => \Laratrust\Middleware\LaratrustAbility::class,
   ```

Configuration

After Installation

Configuration Files

Laratrust now allows mutliple user models, so in order to configure it correctly, you must change the values inside the `config/laratrust.php` file.

Multiple User Models

Inside the `config/laratrust.php` file you will find an `user_models` array, it contains the information about the multiple user models and the name of the relationships inside the `Role` and `Permission` models. For example:

```
'user_models' => [
    'users' => 'App\User',
],
```

Note: The value of the `key` inside the `key => value` pair defines the name of the relationship inside the `Role` and `Permission` models.

It means that there is only one user model using Laratrust, and the relationship with the `Role` and `Permission` models is going to be called like this:

```
$role->users;
$role->users();
```

Note: Inside the `role_user` and `permission_user` table the `user_type` column will be set with the user's fully qualified class name, as the polymorphic relations describe it in Laravel docs.

So if you want to use the MorphMap feature just change the `use_morph_map` value to `true` inside Laratrust's configuration file.

Automatic setup (Recommended)

If you want to let laratrust to setup by itselft, just run the following command:

```
php artisan laratrust:setup
```

This command will generate the migrations, create the `Role` and `Permission` models and will add the trait to the configured user models.

Note: The user trait will be added to the Model configured in the `auth.php` file.

And then do not forget to run:

```
composer dump-autoload
```

Important: If you did the steps above you are done with the configuration, if not, please read and follow the whole configuration process

Migrations

Now generate the Laratrust migration:

```
php artisan laratrust:migration
```

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

```
php artisan migrate
```

After the migration, five new tables will be present:

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

Models

Role

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

```
<?php namespace App;

use Laratrust\LaratrustRole;

class Role extends LaratrustRole
{
}
```

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.

Permission

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

```
<?php namespace App;

use Laratrust\LaratrustPermission;

class Permission extends LaratrustPermission
{
}
```

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`.”

User

Next, use the `LaratrustUserTrait` trait in your existing user models. For example:

```
<?php

use Laratrust\Traits\LaratrustUserTrait;
```

```
class User extends Model
{
   use LaratrustUserTrait; // add this trait to your user model

   ...
}
```

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

Do not forget to dump composer autoload:

```
composer dump-autoload
```

Important: At this point you are ready to go

Seeder

Laratrust comes with a database seeder, this seeder helps you filling the permissions for each role depending on the module, and creates one user for each role.

Note: Laratrust now accepts multiple user models so the seeder is going to work with the first user model inside the user_models array.

To generate the seeder you have to run:

```
php artisan laratrust:seeder
```

and:

```
composer dump-autoload
```

And in the `database/seeds/DatabaseSeeder.php` file you have to add this to the `run` method:

```
$this->call(LaratrustSeeder::class);
```

Note: If you **have not** run `php artisan vendor:publish --tag="laratrust"` you should run it in order to customize the roles, modules and permissions in each case.

Your `config/laratrust_seeder.php` file looks like this:

```
return [
   'role_structure' => [
       'superadministrator' => [
           'users' => 'c,r,u,d',
           'acl' => 'c,r,u,d',
           'profile' => 'r,u'
       ],
       'administrator' => [
           'users' => 'c,r,u,d',
           'profile' => 'r,u'
       ],
       'user' => [
            'profile' => 'r,u'
       ],
   ],
   'permission_structure' => [
       'cru_user' => [
           'profile' => 'c,r,u'
       ],
   ],
   ...
];
```

To understand the `role_structure` you must know:

- The first level is the roles.
- The second level is the modules.
- The second level assignments are the permissions.

With that in mind, you should arrange your roles, modules and permissions like this:

```
return [
    'role_structure' => [
        'role' => [
             'module' => 'permissions',
        ],
    ]
];
```

To understand the `permission_structure` you must know:

- The first level is the users.
- The second level is the modules.
- The second level assignments are the permissions.

With that in mind, you should arrange your users, modules and permissions like this:

```
return [
    'permission_structure' => [
        'user' => [
             'module' => 'permissions',
        ],
    ]
];
```

Permissions

In case that you do not want to use the `c,r,u,d` permissions, in the `config/laratrust_seeder.php` there the `permissions_map` where you can change the permissions mapping.

Usage

Concepts

Set things up

Let's start by creating the following Roles:

```
$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();
```

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();
```

Role Permissions Assignment & Removal

By using the LaratrustRoleTrait we can do the following:

Assignment

```
$admin->attachPermission($createPost);
// equivalent to $admin->permissions()->attach([$createPost->id]);

$owner->attachPermissions([$createPost, $editUser]);
// equivalent to $owner->permissions()->attach([$createPost->id, $editUser->id]);

$owner->syncPermissions([$createPost, $editUser]);
// equivalent to $owner->permissions()->sync([$createPost->id, $editUser->id]);
```

Removal

```
$admin->detachPermission($createPost);
// equivalent to $admin->permissions()->detach([$createPost->id]);

$owner->detachPermissions([$createPost, $editUser]);
// equivalent to $owner->permissions()->detach([$createPost->id, $editUser->id]);
```

User Roles Assignment & Removal

With both roles created let's assign them to the users. Thanks to the `LaratrustUserTrait` this is as easy as:

Assignment

```
$user->attachRole($admin); // parameter can be an Role object, array, id or the role␣
↪string name
// equivalent to $user->roles()->attach([$admin->id]);

$user->attachRoles([$admin, $owner]); // parameter can be an Role object, array, id␣
↪or the role string name
// equivalent to $user->roles()->attach([$admin->id, $owner->id]);

$user->syncRoles([$admin->id, $owner->id]);
// equivalent to $user->roles()->sync([$admin->id]);
```

Removal

```
$user->detachRole($admin); // parameter can be an Role object, array, id or the role␣
↪string name
// equivalent to $user->roles()->detach([$admin->id]);

$user->detachRoles([$admin, $owner]); // parameter can be an Role object, array, id␣
↪or the role string name
// equivalent to $user->roles()->detach([$admin->id, $owner->id]);
```

User Permissions Assignment & Removal

You can attach single permissions to an user, so in order to do it you only have to make:

Assignment

```
$user->attachPermission($editUser); // parameter can be an Permission object, array,␣
↪id or the permission string name
// equivalent to $user->permissions()->attach([$editUser->id]);

$user->attachPermissions([$editUser, $createPost]); // parameter can be an Permission␣
↪object, array, id or the permission string name
// equivalent to $user->permissions()->attach([$editUser->id, $createPost->id]);
```

```
$user->syncPermissions([$editUser->id, $createPost->id]);
// equivalent to $user->permissions()->sync([$editUser->id, createPost->id]);
```

Removal

```
$user->detachPermission($createPost); // parameter can be an Permission object, array,
↪ id or the permission string name
// equivalent to $user->roles()->detach([$createPost->id]);

$user->detachPermissions([$createPost, $editUser]); // parameter can be an Permission␣
↪object, array, id or the permission string name
// equivalent to $user->roles()->detach([$createPost->id, $editUser->id]);
```

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
```

Note: If you want, you can use the `hasPermission` and `isAbleTo` methods instead of the `can` method.

Note: If you want to use the Authorizable trait alongside Laratrust please check *Troubleshooting*.

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

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

// is identical to

```

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

Warning: There aren't `Laratrust::hasPermission` or `Laratrust::isAbleTo` facade methods, because you can use the `Laratrust::can` even when using the `Authorizable` trait.

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
```

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.
- `options` is a set of options to change the method behavior.

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

```
$user->ability(['admin', 'owner'], ['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 = [
    '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 = [
    'validate_all' => true,
    'return_type' => 'both'
];

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

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

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

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

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

// is identical to

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

Objects's Ownership

If you need to check if the user owns an object you can use the user function `owns`:

```
public function update (Post $post) {
   if ($user->owns($post)) { //This will check the 'user_id' inside the $post
      abort(403);
   }

   ...
}
```

If you want to change the foreign key name to check for, you can pass a second attribute to the method:

```
public function update (Post $post) {
   if ($user->owns($post, 'idUser')) { //This will check for 'idUser' inside the $post
      abort(403);
   }

   ...
}
```

Permissions, Roles and Ownership Checks

If you want to check if an user can do something or has a role, and also is the owner of an object you can use the `canAndOwns` and `hasRoleAndOwns` methods:

Both methods accept three parameters:

- `permission` or `role` are the permission or role to check (This can be an array of roles or permissions).
- `thing` is the object used to check the ownership .

- `options` is a set of options to change the method behavior (optional).

The third parameter is an options array:

```
$options = [
    'requireAll' => true | false (Default: false),
    'foreignKeyName'  => 'canBeAnyString' (Default: null)
];
```

Here's an example of the usage of both methods:

```
$post = Post::find(1);
$user->canAndOwns('edit-post', $post);
$user->canAndOwns(['edit-post', 'delete-post'], $post);
$user->canAndOwns(['edit-post', 'delete-post'], $post, ['requireAll' => false,
↪'foreignKeyName' => 'writer_id']);

$user->hasRoleAndOwns('admin', $post);
$user->hasRoleAndOwns(['admin', 'writer'], $post);
$user->hasRoleAndOwns(['admin', 'writer'], $post, ['requireAll' => false,
↪'foreignKeyName' => 'writer_id']);
```

The `Laratrust` class has a shortcut to `owns()`, `canAndOwns` and `hasRoleAndOwns` methods for the currently logged in user:

```
Laratrust::owns($post);
Laratrust::owns($post, 'idUser');

Laratrust::canAndOwns('edit-post', $post);
Laratrust::canAndOwns(['edit-post', 'delete-post'], $post, ['requireAll' => false,
↪'foreignKeyName' => 'writer_id']);

Laratrust::hasRoleAndOwns('admin', $post);
Laratrust::hasRoleAndOwns(['admin', 'writer'], $post, ['requireAll' => false,
↪'foreignKeyName' => 'writer_id']);
```

Ownable Interface

If the object ownership is witha a more complex logic you can implement the Ownable interface so you can use the `owns`, `canAndOwns` and `hasRoleAndOwns` methods in these cases:

```
class SomeOwnedObject implements \Laratrust\Contracts\Ownable
{
   ...

   public function ownerKey()
   {
      return $this->someRelationship->user->id;
   }

   ...
}
```

Note: The `ownerKey` method **must** return the object's owner id value.

And then in your code you can simply do:

```
$user = User::find(1);
$theObject = new SomeOwnedObject;
$user->owns($theObject);            // This will return true or false depending of␣
↪what the ownerKey method returns
```

Soft Deleting

The default migration takes advantage of `onDelete('cascade')` clauses within the pivot tables to remove relations when a parent record is deleted. If for some reason you can not use cascading deletes in your database, the LaratrustRole and LaratrustPermission classes, and the HasRole trait include event listeners to manually delete records in relevant pivot tables. In the interest of not accidentally deleting data, the event listeners will **not** delete pivot data if the model uses soft deleting. However, due to limitations in Laravel's event listeners, there is no way to distinguish between a call to `delete()` versus a call to `forceDelete()`. For this reason, **before you force delete a model, you must manually delete any of the relationship data** (unless your pivot tables uses cascading deletes). For example:

```
$role = Role::findOrFail(1); // Pull back a given role

// Regular Delete
$role->delete(); // This will work no matter what

// Force Delete
$role->users()->sync([]); // Delete relationship data
$role->permissions()->sync([]); // Delete relationship data

$role->forceDelete(); // Now force delete will work regardless of whether the pivot␣
↪table has cascading delete
```

Blade Templates

Three directives are available for use within your Blade templates. What you give as the directive arguments will be directly passed to the corresponding `Laratrust` function. :

```
@role('admin')
    <p>This is visible to users with the admin role. Gets translated to
    \Laratrust::hasRole('admin')</p>
@endrole

@permission('manage-admins')
    <p>This is visible to users with the given permissions. Gets translated to
    \Laratrust::can('manage-admins'). The @can directive is already taken by core
    laravel authorization package, hence the @permission directive instead.</p>
@endpermission

@ability('admin,owner', 'create-post,edit-user')
    <p>This is visible to users with the given abilities. Gets translated to
    \Laratrust::ability('admin,owner', 'create-post,edit-user')</p>
@endability

@canAndOwns('edit-post', $post)
    <p>This is visible if the user has the permission and owns the object. Gets␣
↪translated to
    \Laratrust::canAndOwns('edit-post', $post)</p>
@endOwns
```

```
@hasRoleAndOwns('admin', $post)
    <p>This is visible if the user has the role and owns the object. Gets translated␣
↪to
    \Laratrust::hasRoleAndOwns('admin', $post)</p>
@endOwns
```

Middleware

Concepts

You can use a middleware to filter routes and route groups by permission or role:

```
Route::group(['prefix' => 'admin', 'middleware' => ['role:admin']], function() {
    Route::get('/', 'AdminController@welcome');
    Route::get('/manage', ['middleware' => ['permission:manage-admins'], 'uses' =>
↪'AdminController@manageAdmins']);
});
```

It is possible to use pipe symbol as *OR* operator:

```
'middleware' => ['role:admin|root']
```

To emulate *AND* functionality just use multiple instances of middleware:

```
'middleware' => ['role:owner', 'role:writer']
```

For more complex situations use `ability` middleware which accepts 3 parameters: roles, permissions, validate_all:

```
'middleware' => ['ability:admin|owner,create-post|edit-user,true']
```

Middleware Return

The middleware supports two kinds of returns in case the check fails. You can configure the return type and the value in the `config/laratrust.php` file.

Abort

By default the middleware aborts with a code `403` but you can customize it by changing the `middleware_params` value.

Redirect

To make a redirection in case the middleware check fails, you will need to change the `middleware_handling` value to `redirect` and the `middleware_params` to the route you need to be redirected. Leaving the configuration like this:

```
'middleware_handling' => 'redirect',
'middleware_params'   => '/home',       // Change this to the route you need
```

Short Syntax Route Filter

Note: It only works with laravel `<5.1`

To filter a route by permission or role you can call the following in your `app/Http/routes.php`

```
// only users with roles that have the 'manage_posts' permission will be able to␣
↪access any route within admin/post
Laratrust::routeNeedsPermission('admin/post*', 'create-post');

// only owners will have access to routes within admin/advanced
Laratrust::routeNeedsRole('admin/advanced*', 'owner');

// optionally the second parameter can be an array of permissions or roles
// user would need to match all roles or permissions for that route
Laratrust::routeNeedsPermission('admin/post*', array('create-post', 'edit-comment'));
Laratrust::routeNeedsRole('admin/advanced*', array('owner','writer'));
```

Both of these methods accept a third parameter. If the third parameter is null then the return of a prohibited access will be `App::abort(403)`, otherwise the third parameter will be returned.

So you can use it like:

```
Laratrust::routeNeedsRole('admin/advanced*', 'owner', Redirect::to('/home'));
```

Furthermore both of these methods accept a fourth parameter. It defaults to true and checks all roles/permissions given. If you set it to false, the function will only fail if all roles/permissions fail for that user. Useful for admin applications where you want to allow access for multiple groups :

```
// if a user has 'create-post', 'edit-comment', or both they will have access
Laratrust::routeNeedsPermission('admin/post*', array('create-post', 'edit-comment'),␣
↪null, false);

// if a user is a member of 'owner', 'writer', or both they will have access
Laratrust::routeNeedsRole('admin/advanced*', array('owner','writer'), null, false);

// if a user is a member of 'owner', 'writer', or both, or user has 'create-post',
↪'edit-comment' they will have access
// if the 4th parameter is true then the user must be a member of Role and must have␣
↪Permission
Laratrust::routeNeedsRoleOrPermission(
    'admin/advanced*',
    array('owner', 'writer'),
    array('create-post', 'edit-comment'),
    null,
    false
);
```

Troubleshooting

If you encounter an error when doing the migration that looks like:

```
SQLSTATE[HY000]: General error: 1005 Can't create table 'laravelbootstrapstarter.#sql-
↪42c_f8' (errno: 150)
    (SQL: alter table `role_user` add constraint role_user_user_id_foreign foreign␣
↪key (`user_id`)
    references `users` (`id`)) (Bindings: array ())
```

Then it is likely that the `id` column in your user table does not match the `user_id` column in `role_user`. Make sure both are `INT(10)`.

—

When trying to use the LaratrustUserTrait methods, you encounter the error which looks like:

```
Class name must be a valid object or a string
```

Then probably you do not have published Laratrust assets or something went wrong when you did it. First of all check that you have the `laratrust.php` file in your `app/config` directory. If you don't, then try `php artisan vendor:publish` and, if it does not appear, manually copy the `/vendor/santigarcor/laratrust/src/config/config.php` file in your config directory and rename it `laratrust.php`.

—

If you want to use the `Authorizable` trait yo have to do:

```
use Authorizable {
    Authorizable::can insteadof LaratrustUserTrait;
    LaratrustUserTrait::can as laratrustCan;
}
```

And then replace all the uses of `can` with `hasPermission` or `isAbleTo`.

Note: If you use the `Laratrust::can` facade method you don't have to change this method because it calls the `hasPermission` method.

—

License

Laratrust is free software distributed under the terms of the MIT license.

Contributing

Support follows `PSR-1` and `PSR-4` PHP coding standards, and semantic versioning. Additionally the source code follows the `PSR-2` code style and the builds check it.

Please report any issue you find in the issues page. Pull requests are welcome.