Laratrust Docs Documentation

Release 5.0.0

Santiago Garcia

Aug 14, 2018

Contents

Table of Contents:

Note: Please read all the sections in order.

1.1 Upgrade from 5.0 to 5.1

Important: Laratrust 5.1 requires Laravel >= 5.2.32 and php >= 5.6.0.

In order to upgrade from Laratrust 5.0 to 5.1 you have to follow these steps:

1. Change your `composer.json` to require the 5.1 version of Laratrust:

   ```
   "santigarcor/laratrust": "5.1.*"
   ```

2. Run `composer update` to update the source code.

3. Run `php artisan config:clear` and `php artisan cache:clear`.

4. Update your `config/laratrust.php`:

   4.1. Backup your `config/laratrust.php` configuration values.

   4.2. Delete the `config/laratrust.php` file.

   4.3. Run `php artisan vendor:publish --tag=laratrust`.

   4.4. Update the `config/laratrust.php` file with your old values.

5. Delete the `LaratrustSeeder.php` file and run `php artisan laratrust:seeder`.

6. Run `composer dump-autoload`.

Now you can use the 5.1 version without any problem.

1.2 Installation

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

```
composer require "santigarcor/laratrust:5.0.*"
```

Note: If you are using Laravel 5.5 you don't need to do steps 2 and 3.

---

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. Publish all the configuration files:

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

Warning: If this command did not publish any files, chances are, the Laratrust service provider hasn't been registered. Try clearing your configuration cache:

```
php artisan config:clear
```

---

5. The *middleware* are registered automatically as `role`, `permission` and `ability`. If you want to customize or change them, please refer to the *middleware configuration*.

1.3 Configuration

1.3.1 After Installation

Configuration Files

In your `config/laratrust.php` file you will find all the package configurations that you can customize.

Teams Feature

If you want to use the teams feature that allows you to attach roles and permissions to an user depending on a team, you must change the `use_teams` key value to `true` in your `config/laratrust.php` file. Then follow the *teams configuration* guide.

Multiple User Models

In 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` in 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` tables the `user_type` column will be set with the user's fully qualified class name, as the [polymorphic](#) relations describe it in Laravel docs.

If you want to use the MorphMap feature just change the `use_morph_map` value to `true` in Laratrust's configuration file.

Automatic setup (Recommended)

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

```
php artisan laratrust:setup
```

Warning: If Laravel does not recognize this command, the Laratrust service provider hasn't been registered. Check `providers` array in `config/app.php` and try clearing your configuration cache:

```
php artisan config:clear
```

This command will generate the migrations, create the `Role` and `Permission` models (if you are using the teams feature it will also create a `Team` model) and will add the trait to the configured user models.

Note: The user trait will be added to the models configured in the `config/laratrust.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

1.3.2 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 (or six if you use teams feature) new tables will be present:

- `roles` — stores role records.
- `permissions` — stores permission records.
- `teams` — stores teams records (Only if you use the teams feature).
- `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.

1.3.3 Models

Role

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

```php
<?php

namespace App;

use Laratrust\Models\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
<?php

namespace App;

use Laratrust\Models\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`."

Team

Important: Only applies if you are using the teams feature.

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

```php
<?php

namespace App;

use Laratrust\Models\LaratrustTeam;

class Team extends LaratrustTeam
{
}
```

The `Team` model has three main attributes:

- `name` — Unique name for the Team, used for looking up team information in the application layer. For example: "my-team", "my-company".

- `display_name` — Human readable name for the Team. Not necessarily unique and optional. For example: "My Team", "My Company".

- `description` — A more detailed explanation of what the Team does. Also, optional.

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

User

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

```php
<?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)`, `ability($roles, $permissions, $options)`, and `rolesTeams()` within your `User` model.

Do not forget to dump composer autoload:

```
composer dump-autoload
```

Important: At this point you are ready to go

1.3.4 Seeder

Laratrust comes with a database seeder, this seeder helps you fill 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.

Note: Laratrust now has teams feature, the seeder doesn't support it.

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.

1.3.5 Teams

Note: The teams feature is **optional**, this part covers how to configure it after the installation.

If you had your `use_teams` set to `true` during the installation and automatic setup, you can skip this part.

1. Set the `use_teams` value to `true` in your `config/laratrust.php` file.

2. Run `php artisan laratrust:setup-teams`.

3. Run `php artisan migrate` to apply the changes to the database.

Now you can use the teams feature in you code.

Please refer to the *Teams* concepts to learn how to use them.

1.4 Usage

1.4.1 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 need to add `Permissions`:

```
$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); // parameter can be a Permission object, array␣
→or id
// equivalent to $admin->permissions()->attach([$createPost->id]);

$owner->attachPermissions([$createPost, $editUser]); // parameter can be a Permission␣
→object, array or id
// equivalent to $owner->permissions()->attach([$createPost->id, $editUser->id]);

$owner->syncPermissions([$createPost, $editUser]); // parameter can be a Permission␣
→object, array or id
// equivalent to $owner->permissions()->sync([$createPost->id, $editUser->id]);
```

Removal

```
$admin->detachPermission($createPost); // parameter can be a Permission object, array␣
→or id
// equivalent to $admin->permissions()->detach([$createPost->id]);

$owner->detachPermissions([$createPost, $editUser]); // parameter can be a Permission␣
→object, array or id
// 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 a Role object, array, id or the role␣
→string name
// equivalent to $user->roles()->attach([$admin->id]);

$user->attachRoles([$admin, $owner]); // parameter can be a 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, $owner->id]);

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

Removal

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

$user->detachRoles([$admin, $owner]); // parameter can be a 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 a user, so in order to do it you only have to make:

Assignment

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

$user->attachPermissions([$editUser, $createPost]); // parameter can be a 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]);

$user->syncPermissionsWithoutDetaching([$editUser, $createPost]); // parameter can be
↪a Permission object, array or id
// equivalent to $user->permissions()->syncWithoutDetaching([$createPost->id,
↪$editUser->id]);
```

Removal

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

$user->detachPermissions([$createPost, $editUser]); // parameter can be a 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.

- If you want, you can use the `isA` and `isAn` methods instead of the `hasRole` method.

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

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

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

$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. Also, you can have as many direct `Permissions`s 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 wildcard to check any matching permission by doing:

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

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

Magic can method

You can check if a user has some permissions by using the magic can method:

```
$user->canCreateUsers();
// Same as $user->can('create-users');
```

If you want to change the case used when checking for the permission, you can change the `magic_can_method_case` value in your `config/laratrust.php` file.

```
// config/laratrust.php
'magic_can_method_case' => 'snake_case', // The default value is 'kebab_case'

// In you controller
$user->canCreateUsers();
// Same as $user->can('create_users');
```

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 pipe separated string or an 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');
```

Retrieving Relationships

The `LaratrustUserTrait` has the `roles` and `permissions` relationship, that return a `MorphToMany` relationships.

The `roles` relationship has all the roles attached to the user.

The `permissions` relationship has all the direct permissions attached to the user.

If you want to retrieve all the user permissions, you can use the `allPermissions` method. It returns a unified collection with all the permissions related to the user (via the roles and permissions relationships).

```
dump($user->allPermissions());
/*
 Illuminate\Database\Eloquent\Collection {#646
  #items: array:2 [
    0 => App\Permission {#662
      ...
      #attributes: array:6 [
        "id" => "1"
        "name" => "edit-users"
        "display_name" => "Edit Users"
        "description" => null
        "created_at" => "2017-06-19 04:58:30"
        "updated_at" => "2017-06-19 04:58:30"
      ]
      ...
    }
    1 => App\Permission {#667
      ...
      #attributes: array:6 [
```

```
            "id" => "2"
            "name" => "manage-users"
            "display_name" => "Manage Users"
            "description" => null
            "created_at" => "2017-06-19 04:58:30"
            "updated_at" => "2017-06-19 04:58:30"
        ]
        ...
      }
    ]
}
*/
```

If you want to retrieve the users that have some role you can use the query scope `whereRoleIs` or `orWhereRoleIs`:

```
// This will return the users with 'admin' role.
$users = User::whereRoleIs('admin')->get();
```

Also, if you want to retrieve the users that have some permission you can use the query scope `wherePermissionIs` or `orWherePermissionIs`:

```
// This will return the users with 'edit-user' permission.
$users = User::wherePermissionIs('edit-user')->get();
```

Objects 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 & Ownership Checks

If you want to check if a 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 resolved through a more complex logic you can implement the Ownable interface so you can use the `owns`, `canAndOwns` and `hasRoleAndOwns` methods in those cases:

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

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

    ...
}
```

Important:

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

- The `ownerKey` method receives as a parameter the object that called the `owns` method.

After implementing it, you can simply do:

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

Teams

Note: The teams feature is **optional**, please go to the *teams configuration* in order to use the feature.

Roles Assignment & Removal

The roles assignment and removal are the same, but this time you can pass the team as an optional parameter.

```
$team = Team::where('name', 'my-awesome-team')->first();
$admin = Role::where('name', 'admin')->first();

$user->attachRole($admin, $team); // parameter can be an object, array, id or the
↪string name.
```

This will attach the `admin` role to the user but only within the `my-awesome-team` team.

You can also attach multiple roles to the user within a team:

```
$team = Team::where('name', 'my-awesome-team')->first();
$admin = Role::where('name', 'admin')->first();
$owner = Role::where('name', 'owner')->first();

$user->attachRoles([$admin, $owner], $team); // parameter can be an object, array, id
↪or the string name.
```

To remove the roles you can do:

```
$user->detachRole($admin, $team); // parameter can be an object, array, id or the
↪string name.
$user->detachRoles([$admin, $owner], $team); // parameter can be an object, array, id
↪or the string name.
```

You can also sync roles within a group:

```
$user->syncRoles([$admin, $owner], $team); // parameter can be an object, array, id
↪or the string name.
```

Important: It will sync the roles depending of the team passed, because there is a `wherePivot` constraint in the syncing method. So if you pass a team with id of 1, it will sync all the roles that are attached to the user where the team id is 1.

So if you don't pass any team, it will sync the roles where the team id is `null` in the pivot table.

Permissions Assignment & Removal

The permissions assignment and removal are the same, but this time you can pass the team as an optional parameter.

```php
$team = Team::where('name', 'my-awesome-team')->first();
$editUser = Permission::where('name', 'edit-user')->first();

$user->attachPermission($editUser, $team); // parameter can be an object, array, id
↪or the string name.
```

This will attach the `edit-user` permission to the user but only within the `my-awesome-team` team.

You can also attach multiple permissions to the user within a team:

```php
$team = Team::where('name', 'my-awesome-team')->first();
$editUser = Permission::where('name', 'edit-user')->first();
$manageUsers = Permission::where('name', 'manage-users')->first();

$user->attachPermissions([$editUser, $manageUsers], $team); // parameter can be an
↪object, array, id or the string name.
```

To remove the permissions you can do:

```php
$user->detachPermission($editUser, $team); // parameter can be an object, array, id
↪or the string name.
$user->detachPermissions([$editUser, $manageUsers], $team); // parameter can be an
↪object, array, id or the string name.
```

You can also sync permissions within a group:

```php
$user->syncPermissions([$editUser, $manageUsers], $team); // parameter can be an
↪object, array, id or the string name.
```

Important: It will sync the permissions depending of the team passed, because there is a `wherePivot` constraint in the syncing method. So if you pass a team with id of 1, it will sync all the permissions that are attached to the user where the team id is 1 in the pivot table.

So if you don't pass any team, it will sync the permissions where the team id is `null` in the pivot table.

Checking Roles & Permissions

The roles and permissions verification is the same, but this time you can pass the team parameter.

The teams roles and permissions check can be configured by changing the `teams_strict_check` value inside the `config/laratrust.php` file. This value can be `true` or `false`:

- If `teams_strict_check` is set to `false`:

When checking for a role or permission if no team is given, it will check if the user has the role or permission regardless if that role or permissions was attached inside a team.

- If `teams_strict_check` is set to `true`:

    When checking for a role or permission if no team is given, it will check if the user has the role or permission where the team id is null.

Check roles:

```
$user->hasRole('admin', 'my-awesome-team');
$user->hasRole(['admin', 'user'], 'my-awesome-team', true);
```

Check permissions:

```
$user->can('edit-user', 'my-awesome-team');
$user->can(['edit-user', 'manage-users'], 'my-awesome-team', true);
```

User Ability

The user ability is the same, but this time you can pass the team parameter.

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

$user->ability(['admin'], ['edit-user'], 'my-awesome-team');
$user->ability(['admin'], ['edit-user'], 'my-awesome-team', $options);
```

Permissions, Roles & Ownership Checks

The permissions, roles and ownership checks work the same, but this time you can pass the team in the options array.

```
$options = [
    'team' => 'my-awesome-team',
    'requireAll' => false,
    'foreignKeyName' => 'writer_id'
];

$post = Post::find(1);
$user->canAndOwns(['edit-post', 'delete-post'], $post, $options);
$user->hasRoleAndOwns(['admin', 'writer'], $post, $options);
```

1.4.2 Events

Laratrust comes with an events system that works like the Laravel model events. The events that you can listen to are **roleAttached**, **roleDetached**, **permissionAttached**, **permissionDetached**, **roleSynced**, **permissionSynced**.

Note: Inside the Role model only the **permissionAttached**, **permissionDetached** and **permissionSynced** events will be fired.

If you want to listen to a Laratrust event, inside your `User` or `Role` models put this:

```php
<?php

namespace App;

use Laratrust\Traits\LaratrustUserTrait;

class User extends Model
{
    use LaratrustUserTrait;

    public static function boot() {
        parent::boot();

        static::roleAttached(function($user, $role, $team) {
        });
        static::roleSynced(function($user, $changes, $team) {
        });
    }
}
```

Note: The $team parameter is optional and if you are not using teams, it will be set to null.

The eventing system also supports observable classes:

```php
<?php

namespace App\Observers;

use App\User;

class UserObserver
{

    public function roleAttached(User $user, $role, $team)
    {
        //
    }

    public function roleSynced(User $user, $changes, $team)
    {
        //
    }
}
```

To register an observer, use the laratrustObserve method on the model you wish to observe. You may register observers in the boot method of one of your service providers. In this example, we'll register the observer in the AppService-Provider:

```php
<?php

namespace App\Providers;

use App\User;
use App\Observers\UserObserver;
use Illuminate\Support\ServiceProvider;
```

```
class AppServiceProvider extends ServiceProvider
{

    public function boot()
    {
        User::laratrustObserve(UserObserver::class);
    }


    ...
}
```

Note: Inside your observable classes you can have your normal model events methods alongside Laratrust's events methods.

Available Events

User Events

- **roleAttached($user, $role, $team = null)**

    – $user: The user to whom the role was attached.

    – $role: The role id that was attached to the $user.

    – $team: The team id that was used to attach the role to the $user.

- **roleDetached($user, $role, $team = null)**

    – $user: The user to whom the role was detached.

    – $role: The role id that was detached from the $user.

    – $team: The team id that was used to detach the role from the $user.

- **permissionAttached($user, $permission, $team = null)**

    – $user: The user to whom the permission was attached.

    – $permission: The permission id that was attached to the $user.

    – $team: The team id that was used to attach the permission to the $user.

- **permissionDetached($user, $permission, $team = null)**

    – $user: The user to whom the permission was detached.

    – $permission: The permission id that was detached from the $user.

    – $team: The team id that was used to detach the permission from the $user.

- **roleSynced($user, $changes, $team)**

    – $user: The user to whom the roles were synced.

    – $changes: The value returned by the eloquent sync method containing the changes made in the database.

    – $team: The team id that was used to sync the roles to the user.

- **permissionSynced()**

- `$user`: The user to whom the permissions were synced.

- `$changes`: The value returned by the eloquent `sync` method containing the changes made in the database.

- `$team`: The team id that was used to sync the permissions to the user.

Role Events

- **permissionAttached($role, $permission)**

  - `$role`: The role to whom the permission was attached.

  - `$permission`: The permission id that was attached to the `$role`.

- **permissionDetached($role, $permission)**

  - `$role`: The role to whom the permission was detached.

  - `$permission`: The permission id that was detached from the `$role`.

- **permissionSynced()**

  - `$role`: The role to whom the permissions were synced.

  - `$changes`: The value returned by the eloquent `sync` method containing the changes made in the database.

1.4.3 Middleware

Configuration

The middleware are registered automatically as `role`, `permission` and `ability`. If you want to change or customize them, go to your `config/laratrust.php` and set the `middleware.register` value to `false` and add the following to the `routeMiddleware` array in `app/Http/Kernel.php`:

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

Concepts

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

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

If you use the pipe symbol it will be an *OR* operation:

```
'middleware' => ['role:admin|root']
// $user->hasRole(['admin', 'root']);


'middleware' => ['permission:edit-post|edit-user']
// $user->hasRole(['edit-post', 'edit-user']);
```

To emulate *AND* functionality you can do:

```
'middleware' => ['role:owner|writer,require_all']
// $user->hasRole(['owner', 'writer'], true);


'middleware' => ['permission:edit-post|edit-user,require_all']
// $user->can(['edit-post', 'edit-user'], true);
```

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

```
'middleware' => ['ability:admin|owner,create-post|edit-user,require_all']
// $user->ability(['admin', 'owner'], ['create-post', 'edit-user'], true)
```

If you want yo use a different guard for the user check you can specify it as an option:

```
'middleware' => ['role:owner|writer,require_all|guard:api']
'middleware' => ['permission:edit-post|edit-user,guard:some_new_guard']
'middleware' => ['ability:admin|owner,create-post|edit-user,require_all|guard:web']
```

Teams

If you are using the teams feature and want to use the middleware checking for your teams, you can use:

```
'middleware' => ['role:admin|root,my-awesome-team,require_all']
// $user->hasRole(['admin', 'root'], 'my-awesome-team', true);


'middleware' => ['permission:edit-post|edit-user,my-awesome-team,require_all']
// $user->hasRole(['edit-post', 'edit-user'], 'my-awesome-team', true);


'middleware' => ['ability:admin|owner,create-post|edit-user,my-awesome-team,require_
↪all']
// $user->ability(['admin', 'owner'], ['create-post', 'edit-user'], 'my-awesome-team',
↪ true);
```

Note: The `require_all` and `guard` parameters are optional.

Middleware Return

The middleware supports two types 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.handlers.abort.code` 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.handlers.redirect.url` to the route you need to be redirected. Leaving the configuration like this:

```
'handling' => 'redirect',
'handlers' => [
    'abort' => [
        'code' => 403
    ],
    'redirect' => [
        'url' => ''/home',        // Change this to the route you need
        'message' => [
            'type' => 'error',
            'content' => ''
        ]
    ]
]
```

1.4.4 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
```

1.4.5 Blade Templates

Five 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
```

(continues on next page)

```
@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
```

1.5 Troubleshooting

—

If you make changes directly to the Laratrust tables and when you run your code the changes are not reflected, please clear your application cache using:

```
php artisan cache:clear
```

Remember that Laratrust uses cache in the roles and permissions checks.

—

If you want to use the `Authorizable` trait you 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.

—

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`.

—

1.6 License

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

1.7 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.