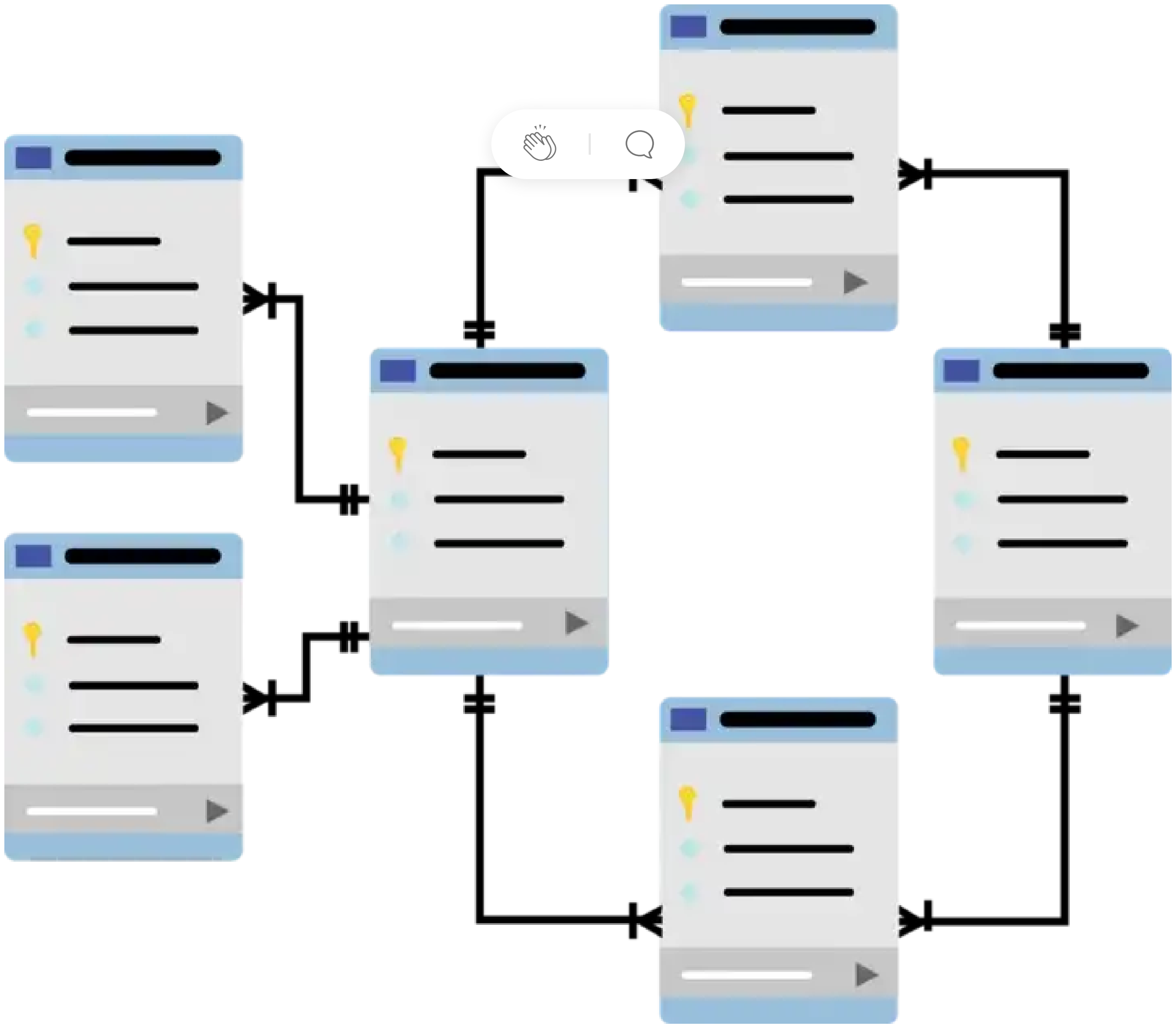
LordNeic Follow

Open in app ↗ Sign up Sign In

#4 Did You Know That Laravel provides an elegant way to handle polymorphic relationships and many-to-many relationships using its Eloquent ORM

Polymorphic relationships allow a model to belong to multiple types of other models on a single association. For example, you might have a `Comment` model that can belong to either a `Post` or a `Video`, depending on the context. This can be achieved using a single `commentable_id` and `commentable_type` column in the `comments` table, where the `commentable_type` column stores the class name of the related model.



Many-to-many relationships, on the other hand, allow many instances of a model to be associated with many instances of another model. For example, a `User` model might have many `Roles`, and a `Role` model might have many `Users`. This relationship is typically implemented using a pivot table that holds the relationships between the two models.

Let's see how we can implement these relationships in Laravel using the `User`, `Permission`, and `Role` example.

Example

To start, let's assume you have a User model set up in your Laravel application. Now let's add few extra models that will support our adventure. These models can be created using the following Artisan command:

```
php artisan make:model Permission -m
php artisan make:model Role -m
php artisan make:model ModelHasRole -m
php artisan make:model ModelHasPermission -m
```

Migrations

Let’s start — here’s an example of migrations for the `roles` and `permissions` tables:

```
        }
    }
```

and here's an example of migrations for the `model_has_role` and `model_has_permission` tables for a polymorphic many-to-many relationship between the Role and Permission models and the User model:

```
        {
            Schema::dropIfExists('model_has_role');
            Schema::dropIfExists('model_has_permission');
        }
    }
```

In these examples, the `model_has_role` and `model_has_permission` tables store the many-to-many relationships between the User model and the Role and Permission models.

Models

Next, let's create the necessary relationships between these models. In the User model, add the following code:

```
public function roles() : BelongsToMany
{
    return $this->belongsToMany(Role::class);
}
public function permissions() : BelongsToMany
{
    return $this->belongsToMany(Permission::class);
}
```

In the Role model, add:

```
public function users() : BelongsToMany
{
    return $this->belongsToMany(User::class);
}
public function permissions() : BelongsToMany
{
    return $this->belongsToMany(Permission::class);
}
```

And in the Permission model:

```
public function users() : BelongsToMany
{
    return $this->belongsToMany(User::class);
}
public function roles() : BelongsToMany
{
    return $this->belongsToMany(Role::class);
}
```

With these relationships set up, we can now create a controller to handle the adding, synchronizing, and removing of permissions and roles from users. For example, let's create a UserController with the following methods:

```
public function addPermission(User $user, Permission $permission)
{
    $user->permissions()->syncWithoutDetaching($permission);
    return redirect()->back()->withSuccess('Permission added successfully');
}

public function removePermission(User $user, Permission $permission)
{
    $user->permissions()->detach($permission);
    return redirect()->back()->withSuccess('Permission removed successfully');
}

public function addRole(User $user, Role $role)
{
    $user->roles()->syncWithoutDetaching($role);
    return redirect()->back()->withSuccess('Role added successfully');
}

public function removeRole(User $user, Role $role)
{
    $user->roles()->detach($role);
    return redirect()->back()->withSuccess('Role removed successfully');
}
```

These methods use the syncWithoutDetaching and detach methods from the Eloquent relationships to add and remove permissions and roles from users.

let's define the routes in our `routes/web.php` file:

```
Route::resource('users', 'UserController');
Route::resource('roles', 'RoleController');
Route::resource('permissions', 'PermissionController');
```

Now, in the `UserController`, we can use the `sync` method to attach or detach Permissions from a User. For example:

```
public function update(Request $request, User $user)
{
    $user->syncPermissions($request->permissions);
    return redirect()->route('users.index');
}
```

In the `RoleController`, we can use the `sync` method to attach or detach Permissions from a Role. For example:

```
public function update(Request $request, Role $role)
{
    $role->syncPermissions($request->permissions);
    return redirect()->route('roles.index');
}
```

Finally, in the `PermissionController`, we can add a `sync` method to attach or detach Roles from a Permission. For example:

```
public function update(Request $request, Permission $permission)
{
    $permission->syncRoles($request->roles);
    return redirect()->route('permissions.index');
}
```

With these methods in place, we can easily manage the relationships between a User, Permission, and Role in Laravel. Whether we need to attach or detach

Permissions from a User, or Roles from a Permission, it's just a matter of calling the appropriate method. This makes it simple to implement complex role-based access control in Laravel applications.

Laravel provides two methods for syncing relationships between models, `sync` and `syncWithoutDetaching`? Understanding the difference between these two methods can help you write more efficient and effective code in Laravel.

Sync vs syncWithoutDetaching

The `sync` method is used to attach or detach relationships to a model. When you call `sync` on a model, any existing relationships not included in the provided array will be detached. Here's an example:

```
$user->syncPermissions([1, 2, 3]);
```

In this example, the `syncPermissions` method attaches the Permissions with the IDs 1, 2, and 3 to the User model. Any Permissions that were previously attached to the User model but not included in the array will be detached.

The `syncWithoutDetaching` method, on the other hand, works similarly to the `sync` method, but it does not detach existing relationships. Here's an example:

```
$user->syncPermissionsWithoutDetaching([1, 2, 3]);
```

In this example, the `syncPermissionsWithoutDetaching` method attaches the Permissions with the IDs 1, 2, and 3 to the User model, but it does not detach any Permissions that were previously attached to the User model but not included in the array.

When deciding which method to use, consider your requirements. If you need to detach relationships, use the `sync` method. If you need to attach relationships without detaching existing ones, use the `syncWithoutDetaching` method. Both methods provide a simple and effective way to manage relationships in Laravel.

Tests

Here's an example of a test for the `syncWithoutDetaching` method for assigning roles and permissions to a user:

```
<?php
namespace Tests\Unit;

use App\Models\Permission;
use App\Models\Role;
use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class UserTest extends TestCase
{
    use RefreshDatabase;
    use WithFaker;

    /**
     * A basic unit test example.
     *
     * @return void
     */
    public function testSyncWithoutDetachingPermissionsAndRoles()
    {
        $user = factory(User::class)->create();
        $permission = factory(Permission::class)->create();
        $role = factory(Role::class)->create();
        $user->syncWithoutDetaching($permission, $role);

        $this->assertTrue($user->hasPermissionTo($permission));
        $this->assertTrue($user->hasRole($role));
    }
}
```

And here's an example of a test for the `sync` method for assigning roles and permissions to a user:

```
<?php
namespace Tests\Unit;

use App\Models\Permission;
use App\Models\Role;
use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class UserTest extends TestCase
{
    use RefreshDatabase;
    use WithFaker;

    /**
     * A basic unit test example.
     *
     * @return void
     */
    public function testSyncPermissionsAndRoles()
    {
        $user = factory(User::class)->create();
        $permission = factory(Permission::class)->create();
        $role = factory(Role::class)->create();
        $user->sync($permission, $role);

        $this->assertTrue($user->hasPermissionTo($permission));
        $this->assertTrue($user->hasRole($role));
    }
}
```

The tests use the `RefreshDatabase` and `WithFaker` traits to reset the database and generate fake data, respectively. The tests use the `syncWithoutDetaching` and `sync` methods to assign a permission and a role to a user, then assert that the user has the expected permission and role assigned to them.

In conclusion, Laravel's Eloquent ORM provides several features for implementing Polymorphism and Many-to-Many relationships in your web application. The usage of Route Resource and Controller Resource helps in managing and organizing routes in a more efficient way. Understanding the difference between sync and syncWithoutDetaching can help you to avoid unexpected data changes and make sure that only the desired data is updated in the database.

Properly defined migrations help to keep your database schema in sync with your application code. Writing tests helps ensure that the application behaves as expected and catches any errors before they cause problems in production.

By using these techniques, you can build a robust, scalable, and maintainable web application using Laravel.

Polymorphism Many To Many Eloquent PHP Laravel