Ressources imbriquées :

Parfois, vous devrez peut-être définir des itinéraires vers une ressource imbriquée. Par exemple, une ressource post peut avoir plusieurs commentaires qui peuvent être joints au post. Pour imbriquer les contrôleurs de ressources, vous pouvez utiliser la notation "point" dans votre déclaration de route :

```
use App\Http\Controllers\PostCommentController;

Route::resource(posts.comments',PostCommentController::class);
```

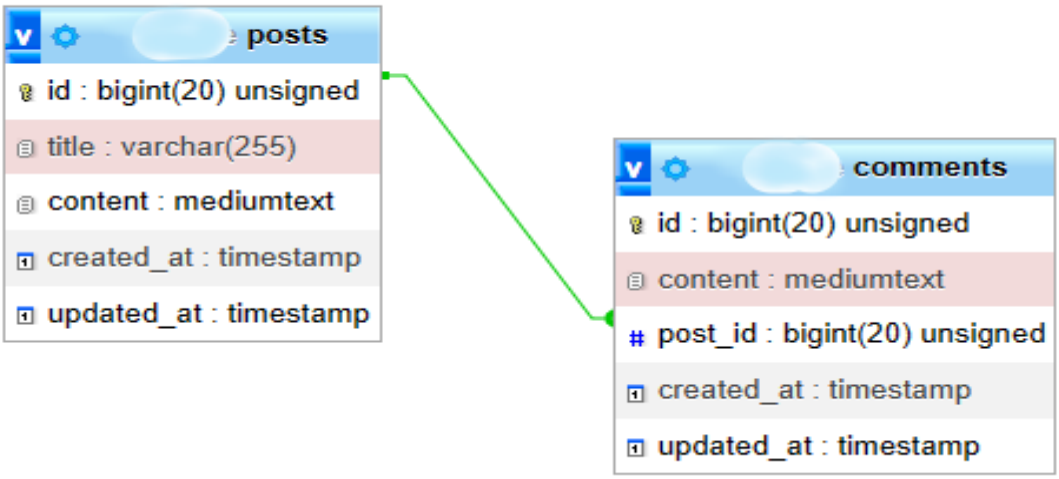
Cette définition de route définira les routes suivantes :

Verb	URI	Action	Route Name
GET	/posts/{post}/comments	index	posts.comments.index
GET	/posts/{post}/comments/create	create	posts.comments.create
POST	/posts/{post}/comments	store	posts.comments.store
GET	/posts/{post}/comments/{comment}	show	posts.comments.show
GET	/posts/{post}/comments/{comment}/edit	edit	posts.comments.edit
PUT	/posts/{post}/comments/{comment}	update	posts.comments.update
DELETE	/posts/{post}/comments/{comment}	destroy	posts.comments.destroy

La méthode **scopeBindings** nous facilitera la vie lorsque nous travaillerons avec des routes qui injectent 2 modèles ayant une relation parent-enfant.

Scénario

Supposons que nous ayons un projet dans lequel les utilisateurs du système peuvent se voir attribuer une ou plusieurs commentaires à un post et dont le schéma de données ressemble à ceci :



posts

	id	title	content	created_at	updated_at
Éditer Copier Supprimer	1	my first post	learn css an javascript	NULL	NULL
Éditer Copier Supprimer	2	my second post	learn python	NULL	NULL

Comments

	id	content	post_id	created_at	updated_at
Éditer Copier Supprimer	1	thanks so much for this course	1	NULL	NULL
Éditer Copier Supprimer	2	goog job dear teacher	1	NULL	NULL

Et dans notre backend nous aurions quelque chose comme :

// app/Models/Post.php

```php
<?php

namespace App\Models;

use App\Models\Comment;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Post extends Model
{
    use HasFactory;
    public function comments(){
        return $this->hasMany(Comment::class);
    }}
```

// app/Models/Comment.php

```php
<?php

namespace App\Models;

use App\Models\Post;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Comment extends Model
{
    use HasFactory;
    public function post(){

     return $this->belongsTo(Post::class);

    }
}
```

//database/migrations/create_posts_table

```php
<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->mediumText('content');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
```

```
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('posts');
    }
};
```

//database/migrations/create_comments_table

```
<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('comments', function (Blueprint $table) {
            $table->id();
            $table->mediumText('content');
            $table->unsignedBigInteger('post_id');
            $table->foreign('post_id')->references('id')->on('posts')-
>onDelete('cascade');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('comments');
    }
};
```

// routes/web.php

```
Route::resource('posts.comments',PostCommentController::class);
```

// app/Http/Controllers/PostCommentController.php

```
    public function show(Post $post,Comment $comment)
    {
       return view('posts.show',compact("comment","post"));
    }

    public function edit(Post $post,Comment $comment)
    {
        return view('posts.edit',compact('post','comment'));
    }
```

// resources/views/posts/edit.blade.php

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>

        <h1>{{$post->title}}</h1>
        <p>{{$comment->content}}</p>


</body>
</html>
```

Supposons maintenant que nous voulons modifier le commentaire avec l'ID 1 du post avec l'ID 1 :

Jusqu'à ce point, tout semble "normal" et fluide, mais si l'URL saisie était :

localhost:8000/posts/2/comments/2/edit

C'est ce que nous aurions fait :

Cela constituerait une incohérence grave pour nos informations car nous éditerions le commenataire d'un autre post. Ce problème se produit parce qu'à aucun moment nous ne validons que l'instance du commentaire est nécessairement un enfant de l'instance du post, nous validons seulement implicitement que chacun existe dans la base de données.

Validation manuelle

Pour s'assurer que seuls les commentaires du post correspondant peuvent être modifiés, nous devons effectuer une validation dans notre contrôleur, qui pourrait par exemple être :

```
    public function edit(Post $post,Comment $comment)
    {
        abort_if($comment->post_id != $post->id, 404);
        return view('posts.edit',compact('post','comment'));
    }
```

Validation avec scopeBingdings

Avec la fonction **scopeBingdings** dans la définition du chemin, nous pouvons dire à Laravel de définir la portée des liaisons "enfants" même si aucune clé personnalisée n'est fournie :

// routes/web.php

```
Route::scopeBindings()->group(function () {

Route::resource('posts.comments',PostCommentController::class);
  });
```

L'utilisation de cette fonction est plus pratique, car elle nous permet de garder notre contrôleur libre de validations supplémentaires lorsque nous utilisons des liens modèle-route dans nos projets.

Localisation des URI de ressource

Par défaut, **Route::resource** créera des URI de ressource en utilisant des verbes anglais et des règles de pluriel. Si vous avez besoin de localiser les verbes d'action create et edit, vous pouvez utiliser la méthode **Route::resourceVerbs**. Cela peut être fait au début de la méthode boot dans votre application **App\Providers\RouteServiceProvider**:

```
public function boot() {
Route::resourceVerbs([
 'create' => 'créer',
 'edit' => 'éditer', ]);
 // ...
}
```

Exemple :

localhost:8000/photos/1/comments/5/éditer

Compléter les contrôleurs de ressources

Si vous devez ajouter des routes supplémentaires à un contrôleur de ressources au-delà de l'ensemble de routes de ressources par défaut, vous devez définir ces routes avant votre appel à la méthode **Route::resource** ; sinon, les routes définies par la méthode **resource** peuvent involontairement prendre le pas sur vos routes supplémentaires:

```
use App\Http\Controller\PhotoController;
Route::get('/student/event', [StudentController::class, 'event']);
Route::resource('student', StudentController::class);
```