AuteurEnHerbe

Zeste de savoir

How to : Développer une API REST avec Laravel pour une Webapp CRUD

4 février 2023

Table des matières

Introduction

NB: une prochaine mise à jour de ce billet apportera des images d'illustration pour alléger le format. Des correctifs au niveau du texte ne sont pas à exclure.

Laravel est un framework PHP open source très populaire et largement utilisé qui a été créé en 2011 par Taylor Otwell. Depuis sa création, il a rapidement gagné en popularité grâce au confort de développement qu'il offre et à ses nombreuses fonctionnalités. Il est particulièrement apprécié par les développeurs pour son architecture et sa documentation claires et bien organisées, ainsi que pour ses nombreux plugins et outils natifs qui facilitent la création de sites web et d'applications web.

De mon constat personnel, la plupart des offres d'emploi backend sont en Laravel, Symfony et Node.js. Selon l'enquête annuelle de Stack Overflow sur les développeurs, Laravel était le cadre de développement web PHP le plus populaire en 2021. Symfony était le deuxième cadre le plus populaire. Selon Google Trends, le nombre de recherches sur Laravel a augmenté de manière significative depuis 2021, tandis que le nombre de recherches sur Symfony a tendance à être plus stable. De plus, une rapide recherche sur Google vous montrera à quel point Laravel dépasse Symfony depuis quelques années, en terme d'utilisation sur les sites Web et API REST. Il en est de même sur SimilarTech.

Dans cet article, je vais vous présenter Laravel et ses principales caractéristiques. Nous verrons au travers d'un projet personnel comment ce framework peut vous aider à développer facilement des applications web, qu'il s'agisse de sites ou d'API REST. Le projet que je vous propose de découvrir en parallèle de mes explications, et qui sert à illustrer les concepts de Laravel que je vous montre, est une API REST pour une Webapp CRUD. Si vous êtes développeur web ou que vous souhaitez découvrir de nouvelles technologies, cet article devrait vous intéresser!

Je vais vous faire découvrir les différentes fonctionnalités de base de Laravel (routes, contrôleurs, *middlewares*, notifications mails, …), l'ORM Eloquent qui facilite la gestion des données, la surcouche logicielle SGBD de Laravel, l'outil d'authentification Sanctum, ainsi que le système de tests disponibles dans Laravel. Nous verrons également comment utiliser les *queues*, *queues workers*, *jobs* et *tasks* pour gérer les tâches en arrière-plan, et comment les fichiers d'environnement et de configuration peuvent être utilisés pour configurer l'application Laravel notamment en fonction du contexte d'exécution du projet (production, tests, développement). Enfin, nous aborderons le système de traductions de Laravel, qui permet de rendre l'application accessible à un public international.

1. Comment j'ai utilisé Laravel pour créer une API REST pour ma webapp CRUD

Mon projet consiste à créer en Vue.js une Webapp communiquant par REST avec un serveur Laravel. Il s'agit d'un *jobs board*, c'est-à-dire un CRM mettant en relation des utilisateurs entreprises publiant des offres d'emploi avec des utilisateurs candidats qui y postulent. Bien sûr, d'autres opérations sont supportées, comme la suppression de jobs, l'envoi d'une notification au candidat si l'entreprise a accepté sa candidature, etc.

Dans ce qui suit, je me sers de ce projet pour illustrer concrètement les concepts et outils Laravel que je vous explique. L'idée est que vous puissiez avoir une idée précise de comment réaliser une API REST pour un projet Web CRUD. Ce projet est accessible sur mon compte GitHub: https://github.com/Jars-of-jam-Scheduler/job-board . Le front en Vue.js est accessible également en tant que *repo* GitHub: https://github.com/Jars-of-jam-Scheduler/job_board_front .

Laravel fournit une API de commandes exécutables dans un terminal pour assister le développeur back-end, que ce soit dans la création des fichiers avec *skeleton* (valable par exemple pour les contrôleurs, tests, etc., liste non-exhaustive) ou dans la gestion du cache (vider le cache de configuration), des données en base (lancer les *seeders*), etc.

La première commande que j'ai saisie est: `curl -s https://laravel.build/the-gummy-bears | bash`, qui va copier/coller depuis ce *repo* distant une version vierge de Laravel dans un répertoire de projet intitulé `the-gummy-bears`. Puis j'ai exécuté ces commandes:

```
cd example-app
./vendor/bin/sail up
```

Laravel Sail est une surcouche Laravel de Docker. Cette commande lance automatiquement tous les *containers Docker* et le réseau nécessaires à l'exécution de l'application Laravel. Laravel Sail et/ou Docker ne sont pas obligatoires, vous pouvez installer Laravel avec d'autres manières de faire ainsi que vous passer de Sail et/ou de Docker: https://laravel.com/docs/9.x .

J'ai par la suite écrit la majorité de mon code à la façon du *tests-driven development*, c'est-à-dire, grossièrement et pour faire simple, par itérations entre l'écriture du code et son test, qui se précisent d'itération en itération. Cela concerne principalement les contrôleurs, modèles Eloquent, système d'authentification Sanctum et notifications mails.

Voici quelques commandes pratiques à taper dans le terminal. Leur signification étant triviale, je vous laisse prendre connaissance de cette liste sans plus tarder. Je vous conseille de lire la suite de mes explications pour savoir pourquoi, quand voire comment les utiliser.

- `php artisan route:list`
- `php artisan make:controller FirmController`
- `php artisan make:controller JobController --resource`
- `php artisan make:resource JobResource`
- `php artisan make:test JobTest`
- `php artisan test --filter JobTest`
- `php artisan route:cache`
- `php artisan make:migration`
- `php artisan make:seed RoleSeeder`
- `php artisan migrate:refresh --seed`
- `php artisan db:seed`
- `php artisan make:factory RoleFactory`
- `php artisan queue:table` (pour créer le fichier de migration qui crée la table des jobs en base: https://laravel.com/docs/9.x/queues#database)
- `php artisan queue:work`

2. Exploration des fonctionnalités de base de Laravel avec l'exemple d'une API REST

Voici une liste des fonctionnalités de base qu'on retrouve dans Laravel:

1. Gestion des routes: Laravel facilite la création et la gestion des routes dans votre application, vous permettant de définir facilement des URLs et de leur associer des actions dans votre code. Ces actions peuvent prendre la forme d'un contrôleur ou d'une fonction PHP anonyme déclarée dans la définition de la route. On peut définir des

vérificateurs de données de formulaire associés aux actions de vos routes. Il est souvent utile de laisser Laravel définir pour nous plusieurs routes permettant de gérer un modèle de données (CRUD) et justement, Laravel fournit un tel outil. Laravel fournit aussi un moyen d'écrire proprement ce qu'un modèle de donnée doit retourner au client lorsque ce dernier le requête à l'aide d'un appel à une route `GET` par exemple.

2. Système de gestion de modèles Eloquent (ORM): Laravel vous permet de définir des modèles pour vos données, vous facilitant la gestion de vos données dans la base de données tout comme leur requêtage. Laravel gère la *soft deletion*.
3. Système d'interaction avec un SGBD: Laravel offre des méthodes générant des requêtes SQL et utilisables conjointement avec Eloquent. Sont aussi disponibles des *migrations* permettant de créer les tables de votre base de données, des *seeders* pour populer cette dernière et des *factories* qui créeront autant d'instances d'un modèle Eloquent que vous le souhaitez en configurant ses données si besoin.
4. Un système d'authentification des utilisateurs, Sanctum, et un système d'autorisation à base de *gates*: Laravel permet de cette façon d'établir par exemple des fonctions de connexion et de déconnexion, mais aussi de les autoriser ou non à appeler telle ou telle route de votre application.
5. Gestion des notifications utilisateurs: Laravel inclut un système permettant d'envoyer des notifications à vos utilisateurs quel que soit le medium (mail, Slack, SMS, Webpush, ...)
6. Système de jobs: Laravel permet de définir des jobs à exécuter en différé de l'exécution de votre application Laravel, ce qui peut être utile notamment dans le cas des notifications comme nous le verrons avec le projet que je vous montre dans ces explications.
7. Définition de tests: il est possible d'écrire des tests pour vous assurer du bon fonctionnement de, disons, vos contrôleurs suite à un appel à une route par exemple.
8. Configuration et Gestion des environnements de travail: Laravel permet de définir des options de configuration utilisables dans votre code et dépendant de l'environnement de travail, qui peut être par exemple, dans le cas le plus simple, un environnement de développement ou un environnement de production.
9. Système de traduction: Laravel comporte un moyen de traduire votre application dans telle ou telle langue.

Nous allons les explorer unes à unes au travers de mon projet CRUD. Eloquent, Sanctum, le système de jobs et le système de tests feront l'objet de parties distinctes de celle que vous lisez actuellement, car je ne les considère pas comme des fonctionnalités dites "de base" en prenant comme référentiel la documentation du framework Laravel.

2.1. Les routes

Laravel permet de définir vos routes dans un fichier:

— routing/web.php
— routing/api.php

Dans mon cas j'ai vidé routing/web.php et rempli routing/api.php, puisque mon projet Laravel est une API REST. Ce fichier contient toutes les routes de votre application qui seront utilisées par l'API. Définir ces routes dans routing/api.php plutôt que dans routing/web.php possède en effet quelques avantages:

— Laravel préfixe automatiquement les URL des routes avec: `/api/`.
— Les routes sont sans état (*i.e.:* elles peuvent être appelées indépendamment de si l'utilisateur final est connecté, contrairement aux routes avec état qui requièrent que l'utilisateur soit connecté). Utile dans certains cas.
— Elles font partie d'un groupe de middlewares[1] spécifique, dont un qui impose une limite sur le nombre d'appels par l'utilisateur final aux routes.

[1]: dans Laravel, un middleware est une classe exécutée avant l'action appelée par une route et qui permet de filtrer ou de modifier la requête.

Voici le contenu de mon routing/api.php:

```php
<?php
use App\Models\User;
use App\Http\Controllers\{UserController, JobController,
    ApplierController, FirmController, EnumController};

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
use Illuminate\Support\Facades\Hash;
use Illuminate\Validation\ValidationException;

/*
|--------------------------------------------------------------------------
| API Routes
|--------------------------------------------------------------------------
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/

Route::post('/sanctum/token', function(Request $request) {
        $request->validate([
                'email' => 'required|email',
                'password' => 'required',
                'device_name' => 'required'
        ]);

        $user = User::where('email', $request->email)->first();
        if(!$user || !Hash::check($request->password,
            $user->password)) {
                throw ValidationException::withMessages([
```

```
31                          'email' => 
                                'The provided credentials are incorrect.'
32                  ]);
33          }
34
35          return $user->createToken($request->device_name)->plainTex
                t;
36  });
37
38  Route::apiResource('jobs', JobController::class);  //
        JobController: Routes are Sanctumed in the controller
39  Route::put('/jobs/{job}/restore', [JobController::class,
        'restore'])->whereNumber('job')->name('jobs_restore');
40
41  Route::middleware('auth:sanctum')->group(function() {
42          /* <!-- Enums --> */
43          Route::get('/enums', [EnumController::class,
                'get'])->name('enums_get');
44
45          /* <!-- User --> */
46          Route::get('/user', [UserController::class,
                'show'])->name('user_show');
47          Route::post('/user/logout', function(Request $request) {
48                  auth()->user()->tokens()->delete();
49          })->name('user_logout');
50
51          /* <!-- Appliers --> */
52          Route::prefix('appliers')->name('appliers.')->group(functi
                on()
                {
53                  Route::put('/', [ApplierController::class,
                        'update'])->name('update');
54                  Route::put('/jobs/{job}/attach',
                        [ApplierController::class, 'attachJob'])->wher
                        eNumber('job')->name('attach_job');
55                  Route::put('/jobs/{job}/detach',
                        [ApplierController::class, 'detachJob'])->wher
                        eNumber('job')->name('detach_job');
56          });
57
58          /* <!-- Firms and Firm --> */
59          Route::prefix('firms')->name('firms.')->group(function() {
60                  Route::put('/', [FirmController::class,
                        'update'])->name('update');
61                  Route::post(
                        '/jobs_applications/{job_application}/accept_or_refuse_job
                        , [FirmController::class,
                        'acceptOrRefuseJobApplication'])->whereNumber(
                        'job_application')->name(
                        'accept_or_refuse_job_application');
```

```
62          });
63
64          Route::prefix('firm')->name('firm.')->group(function() {
65                  Route::get('/jobs', [FirmController::class,
                        'getJobs'])->name('get_jobs');
66                  Route::get('/soft_deleted_jobs',
                        [FirmController::class, 'getSoftDeletedJobs'])
                        ->name('get_soft_deleted_jobs');
67          });
68  });
```

2.1.1. Commentaires

Pour définir une route, vous pouvez utiliser la méthode Route: : suivie de la méthode HTTP souhaitée (par exemple get, post, put, delete) et enfin de l'URL de la route. Vous pouvez également spécifier un nom pour la route en utilisant la méthode name. Ce nom pourra ensuite être utilisé en tant que valeur de paramètre à la fonction `route()` de Laravel, qui permet d'en afficher l'URL.

Il existe de nombreux autres moyens de définir des routes dans Laravel, tels que l'utilisation de groupes de routes (notamment utiles lorsque vous avez plusieurs routes qui partagent un préfixe commun dans l'URL, car cela vous permet de ne pas avoir à répéter ce préfixe pour chaque route individuelle - ou encore pour définir au moins un middleware ou un préfixe de nom de route commun à plusieurs routes, etc.). Un autre concept impliqué dans la définition des routes en Laravel est la méthode `apiResource`, utile pour générer automatiquement un ensemble de routes pour une ressource Eloquent donnée (par exemple: en considérant le modèle Eloquent `Job`, on souhaiterait créer les routes de création, modification, suppression et de listing des jobs sans devoir écrire ces routes unes à unes).

La méthode `whereNumber` que j'utilise dans certaines routes permet d'ajouter une contrainte sur les paramètres de requêtes. Par exemple, il est impossible pour le client d'appeler avec succès l'URL `foobar/api/jobs/abcdef/attach` au lieu de `foobar/api/jobs/99/attach`. Cela permet d'ajouter un niveau de cohérence et de sécurité directement dans la définition de la route plutôt qu'au niveau du contrôleur, avant-même que ce dernier ne soit donc appelé.

Vous pouvez en savoir plus sur la définition des routes dans la documentation de Laravel: https://laravel.com/docs/9.x/routing .

Enfin, un petit mot concernant la toute première route `POST` de mon code, dont l'URL de route est `'/sanctum/token'`, et l'usage du middleware `auth:sanctum`. Cette route est utilisée pour créer un token de connexion (fonctionnalité de Sanctum, un package de Laravel pour les tokens de connexion). Ce token sera par la suite récupéré par le client Vue.js, et renvoyé dans chaque requête nécessitant une session d'authentification (par exemple une requête pour qu'un employeur - donc un utilisateur connecté/authentifié - puisse créer un job). Quant à `auth:sanctum`: la fonction `Route::middleware` est utilisée pour ajouter un middleware à une route ou un groupe de routes. Dans ce cas, le middleware "auth", configuré avec `sanctum` comme authentificateur est utilisé pour protéger toutes la route ou les routes qui suivent dans le groupe, permettant ainsi de s'assurer qu'un utilisateur final qui n'est pas connecté ne pourra pas accéder aux actions associées à ces routes.

2.2. Les contrôleurs

Laravel permet de définir vos contrôleurs dans un fichier sous le namespace `App\Http\Controllers` (compatibilité normes PSR donc les répertoires portent le même nom). Dans mon cas, j'ai écrit 5 contrôleurs:

1. Le contrôleur `EnumController` gère la récupération des valeurs des énumérations PHP (v>=8.1), telles que le type de contrat de travail et l'accord collectif. Ces valeurs seront donc rendues utilisables dans Vue.js pour remplir certaines boîtes de sélection dans les formulaires.
2. Le contrôleur `ApplierController` gère les actions liées aux candidats à un emploi, telles que la mise à jour de leurs informations, le fait de postuler à un emploi et l'annulation d'une candidature.
3. Le contrôleur `FirmController` gère les actions liées aux entreprises, telles que la mise à jour de leurs informations, l'acceptation ou le refus de candidatures à un emploi et la récupération des offres supprimées ou non.
4. Le contrôleur `JobController` gère les actions liées aux offres d'emploi, telles que la publication de nouveaux emplois, la mise à jour d'emplois existants et la suppression d'emplois. Le fait d'y associer des compétences recherchées au niveau du candidat par l'employeur est également implémenté dans la méthode `attachOrDetachJobSkill`.
5. Le contrôleur `UserController` affiche les informations du compte de l'utilisateur connecté.

Voici le contenu de chacun de mes contrôleurs, accompagné de commentaires.

2.2.1. `EnumController`

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Enums\{WorkingPlace, WorkingPlaceCountry,
    EmploymentContractType, CollectiveAgreement};

class EnumController extends Controller
{
    public function get()
        {
                return [
                        'working_place' => collect(WorkingPlace::cases())->map(function ($enum)
                        {
                            return ['value' => $enum->value,
                                'label' => __('working_place.'
                                . $enum->value)];
```

```
15                          }),
16                          'working_place_country' => collect(Working
                                PlaceCountry::cases())->map(function
                                ($enum) {
17                                  return ['value' => $enum->value,
                                        'label' =>
                                        __('working_place_country.' .
                                        $enum->value)];
18                          }),
19                          'employment_contract_type' =>
                                collect(EmploymentContractType::cases(
                                ))->map(function ($enum)
                                {
20                                  return ['value' => $enum->value,
                                        'label' =>
                                        __('employment_contract_type.'
                                        . $enum->value)];
21                                  }),
22                          'collective_agreement' => collect(Collecti
                                veAgreement::cases())->map(function
                                ($enum) {
23                                  return ['value' => $enum->value,
                                        'label' =>
                                        __('collective_agreement.' .
                                        $enum->value)];
24                          }),
25                  ];
26          }
27
28  }
```

Et voici un exemple d'énumération:

```
1   <?php
2   namespace App\Enums;
3
4   enum WorkingPlace : string
5   {
6           case FullRemote = 'full_remote';
7           case HybdridRemote = 'hybrid_remote';
8           case NoRemote = 'no_remote';
9   }
```

2.2.1.1. Commentaires Ce contrôleur de Laravel gère une route HTTP qui renvoie un tableau de valeurs en fonction de différentes énumérations. Le but est de retourner à la fois la valeur brute de l'énumération, par exemple `hybrid_remote`, et la valeur traduite dans la langue

configurée dans Laravel (et pouvant être *reset* celle de l'utilisateur final dans un middleware par exemple).

La méthode `collect` est une méthode de la classe `Illuminate\Support\Collection` qui permet de créer une collection à partir de n'importe quel itérable (par exemple, un tableau). La méthode `map`, de la classe `Collection` justement, permet de parcourir chaque élément de la collection et de le transformer en un autre élément en utilisant une fonction de rappel fournie en argument. Dans ce cas, la fonction de rappel prend en argument chaque énumération et renvoie un tableau associatif contenant la valeur de l'énumération et sa chaîne de caractères localisée ("traduite") associée.

2.2.2. `UserController`

```php
<?php

namespace App\Http\Controllers;

use App\Http\Resources\UserResource;

use Illuminate\Http\Request;

class UserController extends Controller
{
    public function show()
    {
        return new UserResource(auth()->user());
    }
}
```

2.2.2.1. Commentaires sur le contrôleur Il s'agit ici de retourner ce qu'on appelle en Laravel une "API Resource" pour l'utilisateur authentifié (`auth()->user()`). Une *API Resource*, c'est une simple classe qui définit pour un modèle Eloquent donné quels sont les champs de ce dernier qui doivent être retournés au client si celui-ci effectue, par exemple, une requête HTTP `GET`. Il est également possible d'apporter des modifications sur ces champs (par exemple, retourner une traduction au lieu de la valeur brute en base).

Voici d'ailleurs la API Resource `UserResource`:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

```

```php
class UserResource extends JsonResource
{
        /**
     * The "data" wrapper that should be applied.
     *
     * @var string|null
     */
    public static $wrap = 'user';

    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array|\Illuminate\Contracts\Support\Arrayable|\JsonSerializa
     */
    public function toArray($request)
    {
        return [
                        'id' => $this->getKey(),
                        'name' => $this->name,
                        'email' => $this->email,
                        'translated_roles' =>
                            $this->roles->map(function ($role) {
                                return __('roles.' . $role->title);
                        }),
                        'roles' =>  $this->roles->map(function
                            ($role) {
                                return $role->title;
                        }),
                ];
    }
}
```

2.2.2.2. Commentaires sur l'API Resource Lorsque le client fera appel à la route qui est associée à la méthode `show` de ce contrôleur, il recevra les données telles que définies dans cette méthode `toArray` au lieu des données du modèle Eloquent `User` qui auraient sinon été retournées par défaut.

`getKey()` retourne la clé primaire du modèle Eloquent.

Ici, `$wrap`, défini en tant que propriété d'objet, indique à Laravel de remplacer la clé par défaut `data`, qu'il rajoute en amont du tableau JSON contenant les instances du modèle Eloquent récupéré, ou dans notre cas en amont directement de notre instance Eloquent formattée en JSON (puisqu'on ne retourne pas une collection d'*API Resources* mais ben une *API Resource*), par la clé `user`. C'est plus sympa du côté du client Vue.js puisqu'on pourra y accéder avec: `response.data.user;` si on utilise `axios` pour requêter Laravel, au lieu de: `response.data.data`.

2.2.3. `ApplierController`

```php
<?php

namespace App\Http\Controllers;

use App\Models\{Job, User, JobUser,
    AcceptedRefusedJobsApplicationsHistory};
use App\Notifications\{NewJobApplication, AcceptedJobApplication,
    RefusedJobApplication};
use App\Http\Requests\{AttachJobApplierRequest,
    UpdateApplierRequest};

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class ApplierController extends Controller
{
        public function update(UpdateApplierRequest $request)
        {
                $authenticated_user = auth()->user();
                $authenticated_user->fill($request->validated());
                $authenticated_user->update();
                return true;
        }

        public function attachJob(AttachJobApplierRequest
            $request, Job $job)
        {
                $authenticated_user = auth()->user();

                abort_if($authenticated_user->hasAppliedFor($job),
                    400,
                    __('You have already applied for that job.'));

                $authenticated_user->jobs()->attach($job, [
                        'message' => $request->input('message')
                ]);

                $job_application = JobUser::where([
                        [
                                'job_id', '=', $job->getKey()
                        ],
                        [
                                'user_id', '=',
                                    $authenticated_user->getKey()
                        ]
                ])->firstOrFail();
```

```
40                  $job->firm->notify(new
                        NewJobApplication($job_application));
41          }
42
43          public function detachJob(Job $job)
44          {
45                  Gate::authorize('detach-job', $job);
46                  auth()->user()->jobs()->detach($job);
47          }
48  }
```

2.2.3.1. Commentaires sur le contrôleur Ce contrôleur contient trois méthodes: `update`, `attachJob` et `detachJob`.

1. La méthode `update` permet de mettre à jour les informations de l'utilisateur authentifié en utilisant les données soumises dans la requête HTTP et en utilisant la classe `UpdateApplierRequest` pour valider ces données.

`UpdateApplierRequest` est ce qu'on appelle en Laravel un "Form Request". Les *FormRequest* sont des classes de validation de formulaire utilisées dans Laravel pour valider les données envoyées avec des requêtes HTTP. Ils sont généralement utilisés lors de la création ou de la modification de données en utilisant des formulaires HTML (ou JSON dans le cadre d'un appel REST). Lorsqu'une requête HTTP est envoyée au serveur, le `FormRequest` valide les données envoyées avant qu'elles ne soient traitées par le contrôleur. Si les données ne passent pas la validation, le FormRequest renvoie une réponse d'erreur avec les erreurs de validation correspondantes. Si les données sont valides, le contrôleur peut alors traiter les données et effectuer l'action appropriée, comme créer ou mettre à jour un enregistrement en utilisant la méthode `create` ou `fill`.

Voici un exemple de `FormRequest`:

```
1   <?php
2
3   namespace App\Http\Requests;
4
5   use App\Models\Job;
6
7   use Illuminate\Foundation\Http\FormRequest;
8
9   class UpdateApplierRequest extends FormRequest
10  {
11      /**
12       * Determine if the user is authorized to make this request.
13       *
14       * @return bool
15       */
16      public function authorize()
```

```
17      {
18              return true;
19      }
20
21      /**
22       * Get the validation rules that apply to the request.
23       *
24       * @return array<string, mixed>
25       */
26      public function rules()
27      {
28                  return [
29                          'name' => 'nullable|string',
30                  ];
31      }
32  }
```

Voici un autre exemple de `FormRequest::rules`, plus complet:

```
1       /**
2        * Get the validation rules that apply to the request.
3        *
4        * @return array<string, mixed>
5        */
6       public function rules(Request $request)
7       {
8           return [
9               'title' => 'nullable|string',
10                          'firm_id' => 'prohibited',
11                          'presentation' => 'nullable|string',
12                          'min_salary' => 'nullable|integer',
13                          'max_salary' => 'nullable|integer',
14                          'working_place' =>  ['nullable', new
    Enum(WorkingPlace::class)],
15                          'working_place_country' => ['nullable',
    new Enum(WorkingPlaceCountry::class)],
16                          'employment_contract_type' => ['nullable',
    new Enum(EmploymentContractType::class)],
17                          'contractual_working_time' =>
    'nullable|string',
18                          'collective_agreement' => ['nullable', new
    Enum(CollectiveAgreement::class)],
19                          'flexible_hours' => 'nullable|boolean',
20                          'working_hours_modulation_system' =>
    'nullable|boolean',
21
22                          'skill' => 'nullable|array:id,attach_or_de
    tach|required_array_keys:id,attach_or_detach',
```

```
                        'skill.id' => 'integer|gt:0',
                        'skill.attach_or_detach' => 'boolean'
                ];
    }
```

Le champ `skill` de l'exemple ci-dessus peut être omis de la requête, ou, s'il est fourni, doit être un tableau. D'après la règle `array:` que j'ai écrite, seules les clés `id` et `attach_or_detach` seront validées (et donc accessible avec l'appel `$request->validated()` au niveau du contrôleur si toutes ces règles de validation sont passées avec succès par la requête). Si d'autres clés sont présentes dans le tableau, elles feront donc échouer la validation de cette règle (et donc la validation toute entière). La règle `required_array_keys:` indique que ce tableau doit contenir au minimum les clés `id` et `attach_or_detach`. Puis, les règles des clés du tableau `skill`, `id` et `attach_or_detach`, sont également définies.

La méthode `fill` est pratique car elle vous permet de remplir rapidement un modèle avec un grand nombre de données sans avoir à définir chaque propriété du modèle individuellement. Cela peut être particulièrement utile lorsque vous travaillez avec des formulaires HTML ou des requêtes API qui envoient un grand nombre de données. C'est d'ailleurs pourquoi je l'utilise conjointement avec `$request->validated()`, qui retourne uniquement les données validées avec succès par `UpdateApplierRequest::rules`. Dans le cas où la méthode `validated` ne serait pas appelée, et qu'on donnerait directement l'ensemble des données de la requête à la méthode `fill`, il se pourrait que la méthode `update` de mon contrôleur mette à jour des champs envoyés par le client qui ne soient pas désirés (par exemple la clé primaire `id` du modèle `Job`). C'est ce que Laravel appelle "la vulnérabilité de l'assignement de masse". Pour s'assurer que cela n'arrive pas, on peut définir quels sont les attributs qui peuvent être assignés en masse avec la méthode `fill` en définissant par exemple la propriété `protected $fillable` au niveau du modèle:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Casts\Attribute;
use Illuminate\Database\Eloquent\SoftDeletes;

class Job extends Model
{
    use HasFactory, SoftDeletes;

        protected $table = 'firms_jobs';

        protected $fillable = [
                'title',
                'firm_id',
                'presentation',
```

```
                'min_salary',
                'max_salary',
                'working_place',
                'working_place_country',
                'employment_contract_type',
                'contractual_working_time',
                'collective_agreement',
                'flexible_hours',
                'working_hours_modulation_system'
        ];
```

1. La méthode `attachJob` permet à un utilisateur de postuler à un emploi. Si l'utilisateur a déjà postulé pour l'emploi en question, la méthode retourne une erreur HTTP avec un code de statut 400 et un message d'erreur. Sinon, la méthode crée une relation entre l'utilisateur et l'emploi en utilisant la méthode `attach` de la *relationship* `jobs` de l'utilisateur (en ajoutant dans la table pivot le message de candidature), puis envoie une notification `NewJobApplication` à l'entreprise.

`AttachJobApplierRequest` ne se contente pas de valider les données de la requête HTTP reçue par l'API Laravel: elle vérifie également que l'utilisateur final authentifié a le droit de candidater à un job. En effet, une entreprise ne doit pas pouvoir le faire. Il est important de comprendre que cette classe n'a pas pour but de vérifier que l'utilisateur est authentifié: ça, c'est le but du *middleware* `auth:sanctum` défini au niveau des routes qu'il protège de cette façon, comme je l'ai indiqué dans une section antérieure.

Voici le code de `AttachJobApplierRequest`:

```php
<?php

namespace App\Http\Requests;

use App\Models\Job;

use Illuminate\Foundation\Http\FormRequest;

class AttachJobApplierRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
        {
                return $this->user()->can('attach-job');
    }

```

```
21      /**
22       * Get the validation rules that apply to the request.
23       *
24       * @return array<string, mixed>
25       */
26      public function rules()
27      {
28                  return [
29                          'message' => 'string',
30                  ];
31      }
32  }
```

La vérification des droits se passe dans la méthode `authorize`. La méthode `can` vérifie que l'utilisateur authentifié n'a bien pour rôle que `job_applier`, autrement dit qu'il est candidat, en exécutant ce qu'on appelle en Laravel la *gate* `attach-job`. Les *gates* sont utilisées pour vérifier si un utilisateur a une certaine capacité ou permission. Elles peuvent être définies dans votre application, puis appelées depuis vos contrôleurs pour vérifier si un utilisateur est autorisé à effectuer une certaine action. Cela permet de centraliser votre logique d'autorisation et de maintenir votre code propre et organisé.

Voici toutes les *gates* de mon API Laravel (plusieurs sont redondantes et devraient être rendues uniques):

```
1   <?php
2
3   namespace App\Providers;
4
5   use App\Models\{User, Job, JobUser};
6
7   use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
        ServiceProvider;
8   use Illuminate\Support\Facades\Gate;
9
10  class AuthServiceProvider extends ServiceProvider
11  {
12      /**
13       * The model to policy mappings for the application.
14       *
15       * @var array<class-string, class-string>
16       */
17      protected $policies = [
18          // 'App\Models\Model' => 'App\Policies\ModelPolicy',
19      ];
20
21      /**
22       * Register any authentication / authorization services.
23       *
```

```
24          * @return void
25          */
26         public function boot()
27         {
28             $this->registerPolicies();
29 
30                     $this->defineFirmGates();
31                     $this->defineApplierGates();
32         }
33 
34             private function defineFirmGates() : void
35             {
36                     Gate::define('store-job', function(User $user) {
37                             return $user->hasRole('firm') &&
                                    !$user->hasRole('job_applier');
38                     });
39 
40                     Gate::define('destroy-job', function(User $user,
                        Job $job) {
41                             return $user->hasRole('firm') &&
                                    !$user->hasRole('job_applier') &&
                                    $job->firm_id == $user->getKey();
42                     });
43 
44                     Gate::define('update-job-firm', function(User
                        $user, Job $job) {
45                             return $user->hasRole('firm') &&
                                    !$user->hasRole('job_applier') &&
                                    $job->firm_id == $user->getKey();
46                     });
47 
48                     Gate::define('restore-job-firm', function(User
                        $user, Job $job) {
49                             return $user->hasRole('firm') &&
                                    !$user->hasRole('job_applier') &&
                                    $job->firm_id == $user->getKey();
50                     });
51 
52                     Gate::define('accept-or-refuse-job-application',
                        function(User $user, JobUser $job_application)
                        {
53                             return $user->hasRole('firm') &&
                                    !$user->hasRole('job_applier') &&
                                    $job_application->job->firm_id ==
                                    $user->getKey();
54                     });
55 
56                     Gate::define('only-firm', function(User $user) {
57                             return $user->hasRole('firm') &&
                                    !$user->hasRole('job_applier');
```

```
58                  });
59          }
60
61          private function defineApplierGates() : void
62          {
63                  Gate::define('attach-job', function(User $user) {
64                          return $user->hasRole('job_applier') &&
                                !$user->hasRole('firm');
65                  });
66
67                  Gate::define('detach-job', function(User $user,
                        Job $job) {
68                          return $user->hasRole('job_applier') &&
                                !$user->hasRole('firm') &&
                                $job->users()->where('user_id',
                                $user->getKey())->exists();
69                  });
70          }
71  }
```

Une autre chose intéressante dans ce contrôleur, c'est que j'ai défini le paramètre de méthode `Job $job` dans la signature de la méthode `attachJob`. Ce qu'il faut comprendre, c'est que la route `Route::put('/jobs/{job}/attach', [ApplierController::class, 'attachJob'])->whereNumber('job')->name('attach_job');` a besoin d'un nombre, `job`, dans l'URL pour pouvoir être appelée avec succès. Quand le client lui en passe un, Laravel le fournit automatiquement à la méthode `attachJob` du contrôleur `ApplierController` en se basant sur ce même identificateur `job` au niveau de la signature de cette méthode. De plus, comme le paramètre `$job` est *type-hinted* par la classe `Job` qui est un modèle Eloquent, Laravel va automatiquement fournit à la méthode `attachJob` non pas un simple scalaire mais directement une instance de ce modèle, récupérée depuis la base de données. C'est ainsi que Laravel ferait par exemple un appel à `attachJob` en lui passant la valeur `$job` d'ID `99` (modèle Eloquent retrouvé en base de données) dans le cas où le client appellerait cette route avec: `foobar/api/jobs/99/attach`. C'est ce qu'on appelle en Laravel: *route model implicit binding*.

La méthode `hasAppliedFor` est fonction que j'ai moi-même définie dans le modèle Eloquent `User` mais Eloquent fait l'objet d'une section à-part dans mes explications.

Laravel met à disposition plusieurs méthodes pour interagir avec la base de données, qui peuvent être utilisées conjointement à Eloquent, c'est-à-dire sur une instance d'un modèle Eloquent. La fonction `where` est utilisée pour filtrer les enregistrements d'une table de base de données en fonction de certaines conditions. Dans ce cas, elle est utilisée pour filtrer les enregistrements de la table pivot `JobUser` (qui associe les jobs et les utilisateurs, et qui est un modèle Eloquent - en Laravel, on peut définir un modèle ORM Eloquent pour une table pivot, ça peut être pratique pour faire certaines choses) en fonction de deux conditions: la valeur de la colonne `job_id` doit être égale à la valeur de la clé primaire de l'objet `$job` et la valeur de la colonne `user_id` doit être égale à la valeur de la clé primaire de l'objet `$authenticated_user`. La fonction `firstOrFail` est utilisée pour récupérer le premier enregistrement qui correspond

aux conditions spécifiées par la fonction `where`. Si aucun enregistrement ne correspond aux conditions spécifiées, une exception sera lancée.

La méthode `notify` est appelée sur l'objet `firm` associé à l'objet `$job` par *relationship* Eloquent. Cette méthode est donc utilisée pour envoyer une notification à l'entreprise associée au job. L'objet `firm` étant un `User` qui utilise le *trait* PHP `Notifiable` pour pouvoir être notifié. La notification est définie par l'objet `NewJobApplication` qui est passé en argument à la méthode `notify`.

Voici ce que j'ai écrit pour implémenter la notification `NewJobApplication` (bien sûr, les mails ne sont pas configurés pour être réellement envoyés, l'environnement de travail utilisé durant mes tests de développement étant… celui de développement):

```php
<?php

namespace App\Notifications;

use App\Models\JobUser;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;
use Illuminate\Notifications\Notification;

class NewJobApplication extends Notification implements ShouldQueue
{
    use Queueable;

    /**
     * Create a new notification instance.
     *
     * @return void
     */
    public function __construct(private JobUser $job_application)
    {
        //
    }

        public function getJobApplication() : Jobuser
        {
                return $this->job_application;
        }

    /**
     * Get the notification's delivery channels.
     *
     * @param  mixed  $notifiable
     * @return array
     */
```

```
37      public function via($notifiable)
38      {
39          return ['mail'];
40      }
41
42      /**
43       * Get the mail representation of the notification.
44       *
45       * @param  mixed  $notifiable
46       * @return \Illuminate\Notifications\Messages\MailMessage
47       */
48      public function toMail($notifiable)
49      {
50          return (new MailMessage)->view(
51                          'emails.new_job_application',
52                          [
53                                  'applier_name' => $this->job_appli
                                        cation->user->name,
54                                  'job_title' =>
                                        $this->job_application->job
55                          ]
56                  )
57                  ->from('thegummybears@example.fr', 'TheGummyBears')
58                  ->subject('Someone applied to one of your jobs!');
59      }
60
61      /**
62       * Get the array representation of the notification.
63       *
64       * @param  mixed  $notifiable
65       * @return array
66       */
67      public function toArray($notifiable)
68      {
69          return [
70              //
71          ];
72      }
73  }
```

Cette classe de notification implémente l'interface `ShouldQueue`, ce qui signifie qu'elle peut être mise en file d'attente et envoyée plus tard plutôt que d'être envoyée immédiatement. Il suffira de lancer un *queue worker* pour traiter ces envois. Je couvre les concepts de *queue*, *queue worker* et *jobs* dans une autre section.

La classe `NewJobApplication` définit un constructeur qui prend en argument un objet JobUser et le stocke dans une propriété privée `$job_application` par promotion de constructeur (PHP v>=8.0). C'est ce qui permet au contrôleur `ApplierController` de lui fournir l'objet correspondant à la candidature puisque le mail aura besoin de certaines de ses informations dans son texte.

La classe définit également une méthode `via` qui indique que la notification doit être envoyée par courrier électronique (en Laravel, d'autres *media* sont possibles). La méthode `toMail` définit le contenu de l'email qui sera envoyé lorsque la notification est envoyée. Dans cet exemple, l'email sera généré à partir d'une vue de modèle de courrier électronique appelée `emails.new_job_application` et contiendra les variables `applier_name` et `job_title`. L'email sera également envoyé à partir de l'adresse `thegummybears@example.fr` et aura comme objet "Someone applied to one of your jobs !".

Voici ci-dessous le contenu du fichier `new_job_application.blade.php`. Blade est un moteur de template fourni avec Laravel. A noter qu'`emails`, dans `emails.new_job_application`, désigne le répertoire `emails` dans lequel le fichier `new_job_application.blade.php` est contenu.

```
Hello,

We have good news for your job: "{{ $job_title }}"!\n

Do not hesitate to read the message {{ $applier_name }} has
    written to you by logging in on site. We are sure you will
    find your best job here!

\n\n
Best regards,
The Team
```

Blade remplacera `$job_title` et `$applier_name` par les valeurs que nous avons vues définies dans `toMail` de la classe de notification `NewJobApplication`.

1. La méthode detachJob permet à un utilisateur de retirer sa candidature pour un emploi en utilisant la méthode `detach` de l'objet `jobs` (*relationship* Eloquent) de l'utilisateur. Avant de retirer la candidature, la méthode utilise la méthode `authorize` d'une *gate* pour vérifier que l'utilisateur a les autorisations nécessaires pour effectuer cette action. Vous trouverez mes explications concernant ces notions plus haut.

2.2.4. FirmController

La plupart des notions nécessaires à l'écriture des contrôleurs de mon projet ont été expliquée au-dessus. Je vous propose de jeter un rapide coup d'œil au contrôleur `FirmController`, qui utilise quelques concepts qui n'ont pas encore fait l'objet d'explications de ma part.

Voici le contenu de `FirmController`:

```
<?php

namespace App\Http\Controllers;

```

```php
use App\Models\{Job, User, JobUser, AcceptedRefusedJobsApplicationsHistory};
use App\Notifications\{NewJobApplication, AcceptedJobApplication, RefusedJobApplication};
use App\Http\Requests\{AcceptOrRefuseJobApplicationUserRequest, UpdateFirmRequest};
use App\Http\Resources\JobResource;

use Illuminate\Support\Facades\Gate;

class FirmController extends Controller
{
        public function update(UpdateFirmRequest $request)
        {
                $authenticated_user = auth()->user();
                $authenticated_user->fill($request->validated());
                $authenticated_user->update();
                return true;
        }

        public function acceptOrRefuseJobApplication(AcceptOrRefuseJobApplicationUserRequest $request, JobUser $job_application) : AcceptedRefusedJobsApplicationsHistory
        {
                $ret = AcceptedRefusedJobsApplicationsHistory::create([
                        'accepted_or_refused' => $request->accept_or_refuse,
                        'firm_message' => $request->firm_message,
                        'job_application_id' => $job_application->getKey()
                ]);

                if($request->accept_or_refuse) {
                        $job_application->user->notify(new AcceptedJobApplication($job_application));
                } else {
                        $job_application->user->notify(new RefusedJobApplication($job_application));
                }

                return $ret;
        }

        public function getJobs()
        {
                Gate::authorize('only-firm');
                return JobResource::collection(auth()->user()->firmJobs()->latest()->simplePaginate(25));
```

```
43          }
44
45          public function getSoftDeletedJobs()
46          {
47                  Gate::authorize('only-firm');
48                  return JobResource::collection(auth()->user()->fir
                        mJobs()->latest()->onlyTrashed()->simplePagina
                        te(25));
49          }
50  }
```

2.2.4.1. Commentaires sur le contrôleur La méthode `create` est similaire à `fill` précédemment expliquée. Contrairement à `fill`, `create` procède elle-même à un enregistrement en base de données.

`$request->accept_or_refuse` indique à Laravel de récupérer la valeur associée à la clé `accept_or_refuse` reçue par la requête HTTP. En Laravel, on appelle ça la "récupération d'une entrée par propriété dynamique". Confronté à `$request->accept_or_refuse`, Laravel tentera de trouver la valeur de `accept_or_refuse` d'abord dans la charge utile de la requête HTTP (mergée avec les paramètres de la requête HTTP) - si Laravel n'y trouve pas son compte, Laravel tentera de trouver `accept_or_refuse` parmi les paramètres de la route (exemple: `/foobar/{accept_or_refuse}` - `'accept_or_refuse'`). Dans mon cas, de par la définition de la route, Laravel ne pourra trouver cette valeur que dans la charge utile de la requête. Bien entendu, la `FormRequest` `AcceptOrRefuseJobApplicationUserRequest` permet de vérifier que cette valeur est bien présente dans la requête, sinon l'API renverra une exception au client:

```
1   <?php
2
3   namespace App\Http\Requests;
4
5   use App\Models\JobUser;
6
7   use Illuminate\Foundation\Http\FormRequest;
8
9   class AcceptOrRefuseJobApplicationUserRequest extends FormRequest
10  {
11      /**
12       * Determine if the user is authorized to make this request.
13       *
14       * @return bool
15       */
16      public function authorize()
17      {
18          return
                $this->user()->can('accept-or-refuse-job-application',
                $this->route()->parameter('job_application'));
```

```
19      }
20
21      /**
22       * Get the validation rules that apply to the request.
23       *
24       * @return array<string, mixed>
25       */
26      public function rules()
27      {
28                  return [
29                          'accept_or_refuse' => 'required|boolean',
30                          'firm_message' => 'required|string',
31          ];
32      }
33  }
```

Enfin, dernier point qui me semble intéressant à détailler. `return JobResource::collection(auth()->user()->firmJobs()->latest()->simplePaginate(25));` retourne les *API resources*, que nous avons expliqué plus haut, au sein d'une collection paginée (25 éléments par page). A noter l'appel `JobResource::collection()`. La méthode de modèle Eloquent `collection` est utilisée pour créer une nouvelle instance de la classe `Illuminate\Database\Eloquent\Collection` qui contient une liste d'objets de modèle Eloquent. La méthode `collection` peut être utile lorsque vous souhaitez travailler avec une collection d'objets de modèle Eloquent de manière plus flexible, car elle fournit une variété de méthodes pour filtrer, transformer et manipuler la collection. Par exemple, vous pouvez utiliser la méthode `map` pour transformer chaque objet de la collection en un autre objet, ou la méthode `filter` pour filtrer la collection en fonction de certaines conditions. De plus, cette méthode permet de retourner au client une donnée facilement itérable. La méthode `latest` permet de trier les résultats par ordre décroissant sur la colonne de date de création configurée par défaut `created_at` (on peut aussi en spécifier une autre, bien sûr).

2.2.5. JobController

`JobController` est un *Resource Controller*, à ne pas confondre avec les *API Resource* que j'ai abordées plus haut. Il s'agit d'un simple contrôleur Laravel qui doit son nom à la façon dont ses routes sont définies. Jetez un coup d'œil au fichier *api/routing.php*:

```
1  Route::apiResource('jobs', JobController::class);  //
       JobController: Routes are Sanctumed in the controller
```

En appelant `Route::apiResource`, j'indique à Laravel de générer automatiquement certaines routes: `index` (pour retourner la liste de tous les jobs), `store` (pour enregistrer en base un job), `update` (pour mettre à jour en base le job paramètre de la route), `destroy` (pour supprimer le job paramètre de la route), alors Laravel générera automatiquement les routes correspondantes. La documentation de Laravel fournit le tableau associant le nom de ces routes aux

méthodes du contrôleur appelées, à l'URI et au type de méthode HTTP ici: https://laravel.com/docs/9.x/controllers#actions-handled-by-resource-controller .

Voici le contenu du contrôleur `JobController`:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests\StoreJobRequest;
use App\Http\Requests\UpdateJobRequest;
use App\Models\{Job, User};
use App\Http\Resources\JobResource;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class JobController extends Controller
{

        public function __construct()
        {
                $this->middleware('auth:sanctum')->except('index');
        }

        public function index()
        {
                $jobs = Job::latest()->simplePaginate(25);
                return JobResource::collection($jobs);
        }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \App\Http\Requests\StoreJobRequest  $request
     * @return \Illuminate\Http\Response
     */
    public function store(StoreJobRequest $request)
    {
        return Job::create(['firm_id' => auth()->user()->id,
            ...$request->validated()]);
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \App\Http\Requests\UpdateJobRequest  $request
     * @param  \App\Models\Job  $job
     * @return \Illuminate\Http\Response
```

```
44      */
45     public function update(UpdateJobRequest $request, Job $job)
46     {
47                 if($request->has('skill')) {
48                         $this->attachOrDetachJobSkill($request,
                               $job);
49                 }
50
51                 $job->fill(['firm_id' => auth()->user()->id,
                       ...$request->validated()]);
52                 $job->update();
53                 return true;
54     }
55
56     /**
57      * Remove the specified resource from storage.
58      *
59      * @param  \App\Models\Job  $job
60      * @return \Illuminate\Http\Response
61      */
62     public function destroy(Job $job)
63     {
64                 Gate::authorize('destroy-job', $job);
65         return $job->delete();
66     }
67
68         public function restore(int $job_id)
69         {
70                 $job = Job::withTrashed()->findOrFail($job_id);
71                 Gate::authorize('restore-job-firm', $job);
72                 return $job->restore();
73         }
74
75         private function attachOrDetachJobSkill(Request $request,
               Job $job)
76         {
77                 if($request->input('skill.attach_or_detach')) {
78                         $job->skills()->attach($request->input(
                               'skill.id'));
79                 } else {
80                         $job->skills()->detach($request->input(
                               'skill.id'));
81                 }
82         }
83 }
```

2.2.5.1. Commentaires sur le contrôleur En Laravel, on peut utiliser le constructeur d'un contrôleur pour y définir un middleware (au lieu de le faire au niveau de la définition des routes). Cela peut être utile dans le cas de notre appel à `Route::apiResource` dans le fichier des

routes, si l'on veut éviter d'appliquer le middleware sur certaines méthodes du contrôleur comme `index`. Ici, c'est bien ce que j'ai voulu faire: le *listing* des jobs ne doit pas être protégé par Sanctum (c'est-à-dire qu'un utilisateur non-authentifié doit pouvoir y accéder). D'où la ligne `$this->middleware('auth:sanctum')->except('index');` dans le constructeur.

`Job::withTrashed()` dans la méthode `restore` permet d'annuler la suppression d'un job ayant été *soft deleted*. Le concept de *soft deletion* est expliqué plus loin dans la section dédiée à Eloquent.

3. Comment Laravel facilite la gestion des données avec Eloquent, son ORM

L'ORM Eloquent est un outil intégré à Laravel qui, comme tout ORM, permet de simplifier les interactions avec une base de données, en utilisant des objets PHP au lieu de requêtes SQL brut. Cela signifie que vous pouvez utiliser des méthodes et des propriétés d'objet pour effectuer des opérations de base de données, telles que la sélection, l'insertion, la mise à jour et la suppression de données, plutôt que d'écrire des requêtes SQL manuellement.

En utilisant Eloquent, vous pouvez également établir des relations entre les modèles, comme une relation "one-to-one", "one-to-many", "many-to-many" afin de naviguer facilement entre les données de différentes tables. Par exemple, un candidat peut avoir candidaté à plusieurs jobs, et un job peut avoir fait l'objet de plusieurs candidatures (une par candidat). Ou encore: un job ne peut avoir été créé que par un seul utilisateur de rôle "entreprise", mais une entreprise peut avoir créé plusieurs jobs.

On utilise Eloquent en créant des modèles Eloquent - un modèle Eloquent par table concernée. Dans mon cas, j'ai créé ces modèles Eloquent:

- `Job`
- `User`
- `Skill`
- `Role`
- `JobUser`: il s'agit comme je l'ai mentionné plus haut d'un modèle Eloquent représentant la table pivot de la relation job<->user. Autrement dit, tel que j'ai écrit mon projet, une instance `jobUser` (= une entrée de la table associée), c'est une candidature d'un utilisateur de rôle "candidat" à un job. Une colonne `message` est présente en plus des deux colonnes de type *foreign keys*. Il s'agit du message que l'utilisateur authentifié laisse lorsqu'il candidate.
- `AcceptedRefusedJobsApplicationsHistory`: il s'agit des enregistrements comme quoi une entreprise donnée a accepté ou a refusé une candidature. Comme elle peut faire cela plusieurs fois (par exemple accepter dans un premier temps, puis finalement refuser une même candidature), cette notion d'historique s'impose.

3.1. *Factories*, `$timestamps` et *relationships*

Voici un exemple de modèle Eloquent, `Skill`:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Skill extends Model
{
    use HasFactory;

        protected $fillable = [
                'title',
        ];

        public $timestamps = false;

        public function jobs()
        {
                return $this->belongsToMany(Job::class);
        }

}
```

Le code écrit ci-dessus indique que ce modèle bénéficie du système de *factories* Laravel grâce à l'utilisation du *trait* PHP `HasFactory`. Les *factories* dans Laravel sont des outils qui permettent de générer des données de test aléatoires pour votre application. Elles sont particulièrement utiles lorsque vous devez créer beaucoup de données de test dans votre base de données pour tester votre application. Vous pouvez également utiliser la méthode `make` de la *factory* pour générer un objet de test sans l'enregistrer dans la base de données. Cela peut être utile si vous avez besoin de créer plusieurs enregistrements de test avec des données légèrement différentes. Les *factories* peuvent aussi être utilisées dans le cadre de *database seeders*, lesquels permettent de populer une base de données.

Voici la *factory* de ce modèle `Skill` (Laravel la retrouve en se basant sur le nom de la classe du modèle, ici `Skill`, suffixé par la chaîne de caractères: `Factory`) et un exemple d'appel à cette *factory* réalisé dans un *seeder*:

```php
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Sk
```

```
 */
class SkillFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition()
    {
        return [
            'title' => implode(' ', fake()->words(2)),
        ];
    }
}
```

Dans `definition`, j'ai défini le champ `title` et sa valeur de test à générer. Voici le *seeder* qui fait appel à cette *factory*:

```
<?php

namespace Database\Seeders;

use App\Models\Skill;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class SkillSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
                Skill::factory()
                ->count(50)
                ->create();
    }
}
```

Ce *seeder* crée donc 50 skills avec des titres aléatoires.

Dans Laravel, l'attribut `$timestamps` est une propriété des classes modèles Eloquent qui détermine si le modèle doit être horodaté ou non. Lorsque `$timestamps` est défini sur true,

Laravel mettra automatiquement à jour les colonnes `created_at` et `updated_at` sur la table de la base de données correspondant au modèle.

En utilisant Eloquent, vous pouvez également établir des relations entre les modèles, comme une relation "many-to-may" (expliquée plus haut), afin de naviguer facilement entre les données de différentes tables. Dans le modèle `Skill`, j'ai défini la *relationship* `jobs` qui retourne la méthode Laravel `belongsToMany`. Il s'agit justement de la méthode à appeler pour implémenter une *relationship many-to-many*. Cela signifie que le modèle actuel (`skill`) peut être demandé pour plusieurs *jobs* , et chaque compétence peut être liée à plusieurs modèles.

Voici un exemple d'appel à cette *relationship* (contrôleur `JobController`):

```
1        private function attachOrDetachJobSkill(Request $request,
    Job $job)
2        {
3                if($request->input('skill.attach_or_detach')) {
4                        $job->skills()->attach($request->input('sk
    ill.id'));
5                } else {
6                        $job->skills()->detach($request->input('sk
    ill.id'));
7                }
8        }
```

Un appel à `$job->skills` retournerait toutes les compétences de ce job. On peut paginer cela si besoin, si l'on appelle `$job->skills()` en tant que méthode (notez les `()`). Exemple: `return JobResource::collection(auth()->user()->firmJobs()->latest()->simplePaginate(25));` (contrôleur `FirmController`). Les méthodes `attach` et `detach` de l'extrait de code ci-dessus permettent d'ajouter ou de supprimer une entrée dans la table pivot mettant en relation les jobs et les compétences, c'est-à-dire d'indiquer à Laravel que telle compétence n'est plus requise pour exercer tel job et que tel job n'a plus besoin de telle compétence pour être exercé (bien entendu, ces deux phrases sont les mêmes).

D'autres méthodes de *relationships* existent sur le même principe. La différence principale étant l'implémentation des *foreign keys* en base de données. Dans le cas de la *relationship many-to-many*, j'ai créé une table pivot. D'autres relations nécessitent par exemple une simple colonne *foreign* key *(*relationship one-to-many*). Pour plus d'informations à ce sujet, je vous invite à lire la documentation de Laravel: https://laravel.com/docs/9.x/eloquent-relationships .

3.2. *Soft-deletion*, `using` (contexte : définition d'une *relationship*), accesseurs et mutateurs

Voici le code que j'ai écrit pour le modèle `Job`:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Casts\Attribute;
use Illuminate\Database\Eloquent\SoftDeletes;

class Job extends Model
{
    use HasFactory, SoftDeletes;

        protected $table = 'firms_jobs';

        protected $fillable = [
                'title',
                'firm_id',
                'presentation',
                'min_salary',
                'max_salary',
                'working_place',
                'working_place_country',
                'employment_contract_type',
                'contractual_working_time',
                'collective_agreement',
                'flexible_hours',
                'working_hours_modulation_system'
        ];

        public function skills()
        {
                return $this->belongsToMany(Skill::class);
        }

        public function users()
        {
                return $this->belongsToMany(User::class)->using(Jo
                    bUser::class);
        }

        protected function title(): Attribute
        {
                return Attribute::make(
                        get: fn ($value) => ucfirst($value)
                );
        }

        public function firm()
```

```
49          {
50                  return $this->belongsTo(User::class);
51          }
52  }
```

Celui-ci utilise le *trait* PHP `SoftDeletes` et bénéficie donc du système de *soft deletion* Laravel. `SoftDeletes` est un *trait* dans Laravel qui permet de marquer des enregistrements de base de données comme ”supprimés” au lieu de les supprimer définitivement. Lorsqu'un enregistrement est ”supprimé”, il n'est pas visible dans les requêtes de base de données normales, mais il n'est pas non plus complètement effacé de la base de données. Cela permet de conserver une trace de l'historique des données et de pouvoir éventuellement ”récupérer” un enregistrement qui a été ”supprimé” par erreur.

Pour utiliser la fonctionnalité de *soft deletion* dans un modèle, vous devez inclure le *trait* `SoftDeletes` dans votre modèle et définir une colonne de type `timestamp` appelée `deleted_at` dans votre table de base de données (vous pouvez pour ce faire utiliser les méthodes d'ajout de colonnes dans le SGBD `softDeletes` et `dropSoftDeletes` dans le fichier de migration). Lorsqu'un enregistrement est ”supprimé”, la valeur de `deleted_at` sera définie sur la date et l'heure actuelles.

Vous pouvez utiliser la méthode `onlyTrashed` pour afficher uniquement les enregistrements ”supprimés” dans une requête, ou la méthode `withTrashed` pour afficher tous les enregistrements, y compris les enregistrements ”supprimés”. La méthode `restore` permet de restaurer l'entrée qui était *soft deleted*. Voici ci-dessous quelques exemples d'appels.

Contrôleur `JobController`:

```
1           public function restore(int $job_id)
2           {
3                   $job = Job::withTrashed()->findOrFail($job_id);
4                   Gate::authorize('restore-job-firm', $job);
5                   return $job->restore();
6           }
```

Contrôleur `FirmController`:

```
1           public function getSoftDeletedJobs()
2           {
3                   Gate::authorize('only-firm');
4                   return JobResource::collection(auth()->user()->fir
    mJobs()->latest()->onlyTrashed()->simplePaginate(25));
5           }
```

Dans le code du modèle `Job` que j'ai fourni, vous pouvez constater l'utilisation de `using`:

```
1        public function users()
2        {
3                return
    $this->belongsToMany(User::class)->using(JobUser::class);
4        }
```

Cela permet de spécifier un modèle Eloquent correspondant au pivot de cette *relationship* (*many-to-many*). Il n'est pas obligatoire de créer un modèle Eloquent représentant une table pivot (ni d'utiliser `using`). Cela peut être utile pour réaliser certaines choses.

Enfin, je souhaiterais attirer votre attention sur l'extrait de code qui suit:

```
1        protected function title(): Attribute
2        {
3                return Attribute::make(
4                        get: fn ($value) => ucfirst($value)
5                );
6        }
```

Cette méthode définit un accesseur (*accessor*), qui permet de modifier la valeur de l'attribut `title` dès que celui-ci est accédé: le client reçoit donc la valeur modifiée. Dans le cadre de mon projet, c'est surtout utile quand le client requête la récupération d'un job, mais cet accesseur sera en fait exécuté par Laravel dès que l'attribut `title` sera accédé d'une manière ou d'une autre (incluant la récupération d'un job par appel à une route associée à la méthode HTTP `GET` telle que celle définie dans le fichier *api/routing.php*). On peut aussi définir des mutateurs (*mutators*), comme c'est le cas dans la documentation de Laravel par exemple: https://laravel.com/docs/9.x/eloquent-mutators#defining-a-mutator .

3.3. `$primaryKey`, `$incrementing`, `$keyType` et indications de clés à Laravel pour une *relationship*

Les rôles des utilisateurs sont pour l'instant au nombre de deux dans mon projet:

- Un utilisateur peut être un candidat (pouvant postuler à un job, annuler sa candidature, etc.)
- Ou bien une entreprise (pouvant CRUD un job, restaurer un job *soft-deleted*, accepter ou refuser une candidature, etc.)

J'ai décidé d'enregistrer ces deux rôles `job_applier` et `firm` directement dans la base de données, dans une table `roles`.

Voici le code que j'ai écrit pour le modèle Eloquent `Role`:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    use HasFactory;

        protected $primaryKey = 'title';
        public $incrementing = false;
    protected $keyType = 'string';

        protected $fillable = [
                'title',
        ];

        public $timestamps = false;

        public function users()
        {
                return $this->belongsToMany(User::class,
                    'role_user', 'role_title', 'user_id');
        }

}
```

La table `roles` n'a pas de colonne de clé primaire ID numérique. Chaque rôle étant unique, c'est directement son intitulé, écrit dans un format d'identificateur (minuscules et caractère d'espacement: '_') qui est la clé primaire ID chaîne de caractères. Et de faire, non-incrémentable. La documentation de Laravel indique qu'il faut penser à spécifier tout cela dans le modèle Eloquent, d'où les lignes suivantes que vous venez sans doute de lire ci-dessus:

```
        protected $primaryKey = 'title';
        public $incrementing = false;
    protected $keyType = 'string';
```

Vous avez sans doute remarqué la ligne `return $this->belongsToMany(User::class, 'role_user', 'role_title', 'user_id');` qui diffère un peu de ce que je vous ai montré jusqu'à présent. En fait, la méthode `belongsToMany` utilise par convention certains nommages pour identifier la table pivot (bien entendu, les utilisateurs et les rôles sont liés au travers d'un pivot), ainsi que les *foreign keys* de cette table pivot. Si le nom de cette table pivot et le nom d'au moins une de ces *foreign keys* ne correspond pas à ce nommage auquel Laravel s'attend par défaut selon sa convention, on peut les spécifier: c'est ce que je fais ici. Cette fonction-

nalité n'est pas réservée à la seule méthode de *relationship* `belongsToMany` mais aux autres également. Voir la documentation de Laravel à ce sujet: https://laravel.com/docs/9.x/eloquent-relationships#defining-relationships .

4. Laravel et sa surcouche logicielle SGBD

Dans ces explications, j'ai présenté Eloquent, l'ORM par défaut de Laravel. Il s'agit déjà d'une surcouche au SGBD. Laravel, entre le SGBD et Eloquent, fournit une autre surcouche au SGBD, un ensemble de méthodes pour insérer, modifier, supprimer, lire des données en base. Je vous ai déjà montré les méthodes `firstOrFail`, `where`, `create` et `latest` plus haut. Bien entendu de nombreuses autres méthodes sont disponibles mais leur utilisation ne s'est pas imposée dans mon projet. Je vous invite à consulter la documentation de Laravel à ce sujet: https://laravel.com/docs/9.x/database (pensez à lire les différents sous-chapitres - "Query Builder", "Pagination", etc.). Enfin, je vous rappelle qu'on peut utiliser conjointement Eloquent et cette surcouche SGBD: je veux dire par là qu'on peut enchaîner des appels de méthodes Eloquent avec des méthodes de cette surcouche SGBD, et même utiliser directement des méthodes de la surcouche SGBD sur un modèle Eloquent (en appel statique de méthode ou en appel de méthode sur objet instancié).

4.1. *Seeders* et *Migrations*

Dans Laravel, les *seeders* et les *migrations* sont des outils qui vous permettent de gérer les données de votre base de données de manière reproductible.

Les *seeders* sont utilisés pour remplir votre base de données avec des données de test ou de démo. Ils peuvent être utilisés pour initialiser votre base de données avec des données de départ, ou pour peupler votre base de données avec des données de test lors de la réalisation de tests automatisés.

Les *migrations*, quant à elles, sont utilisées pour gérer les modifications de votre structure de base de données de manière organisée. Elles vous permettent de créer, modifier ou supprimer des tables et des colonnes de votre base de données de manière organisée, et de garder une trace de toutes ces modifications dans l'historique de votre projet.

4.1.1. Exemples de *seeders*

J'ai activé les *seeders* suivants (contenu de la méthode `run`):

```php
<?php

namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

```

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
                $this->call([
            SkillSeeder::class,
                RoleSeeder::class,
                FirmSeeder::class,
                        JobApplierSeeder::class,
                        JobSeeder::class,
        ]);
    }
}

Voici deux exemples de *seeders* :

```php
<?php

namespace Database\Seeders;

use App\Models\Job;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class JobSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
                Job::factory()
                ->count(100)
                ->create();
    }
}
```
```

```php
<?php

namespace Database\Seeders;

use App\Models\{User, Role};

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\Hash;

class JobApplierSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        User::factory()
                ->hasAttached([Role::findOrFail('job_applier')])
                ->create([
                        'name' => 'My Applier',
                        'email' => 'applier@applier.fr',
                        'password' => Hash::make('azerty'),
                ]);
    }
}
```

`hasAttached` permet de spécifier le rôle de l'utilisateur à la création de l'utilisateur par la *factory*. Cette méthode est à appeler dans le cas d'une relation *many-to-many* (donc avec une table pivot). Remarquez la terminologie de `hasAttached` (`Attached`) utilisée. Elle ressemble fortement à `attach()` que nous avions vu plus haut, et ce n'est pas un hasard: c'est la terminologie utilisée dans le cas des relations *many-to-many* à base de tables pivots.

4.1.2. Exemple de *migrations*

Pour écrire une *migration*, Laravel met à notre disposition diverses méthodes comme `id` pour définir une colonne de clé primaire, `softDeletes` pour créer la colonne `deleted_at` (dans le cadre de la mise en place du système de *soft deletion*), `text` pour créer une colonne de texte, etc. La méthode `up` est censée contenir tous ces appels et permet de créer la table, à l'inverse de la méthode `down`. Laravel permet aussi de créer des clés étrangères (avec un appel de la forme: `foreign(foo)->references(bar)->on(foobar)`). La méthode `unique` permet de définir l'unicité d'une colonne. Enfin, la méthode `drop` supprime la table.

Voici le fichier de *migration* de la table des jobs:

```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('firms_jobs', function (Blueprint $table) {
                        $table->id();
                        $table->softDeletes();

                        $table->unsignedBigInteger('firm_id');
                        $table->foreign('firm_id')->references('id')->on('users');

                        $table->string('title')->nullable();
                        $table->text('presentation')->nullable();
                        $table->integer('min_salary')->nullable();
                        $table->integer('max_salary')->nullable();
                        $table->enum('working_place', ['full_remote', 'hybrid_remote', 'no_remote'])->nullable();
                        $table->enum('working_place_country', ['fr'])->nullable();
                        $table->enum('employment_contract_type', ['cdi', 'cdd'])->nullable();
                        $table->string('contractual_working_time')->nullable();
                        $table->enum('collective_agreement', ['syntec'])->nullable();
                        $table->boolean('flexible_hours')->nullable();
                        $table->boolean('working_hours_modulation_system')->nullable();
                        $table->timestamps();
                    });
    }

    /**
```

```
40          * Reverse the migrations.
41          *
42          * @return void
43          */
44         public function down()
45         {
46                         Schema::drop('firms_jobs');
47         }
48 };
```

5. Comment Sanctum facilite l'authentification des utilisateurs

Il y a plusieurs façons de mettre en place un système d'authentification dans Laravel. Voici quelques exemples:

- Utiliser le système d'authentification intégré de Laravel: Laravel fournit un système d'authentification intégré à base de méthodes (façades Laravel `Auth` et `Session`). Utilisables pour l'authentification par session et cookie de session (dans le cadre d'un formulaire de site Web par exemple).
- Utiliser un package tiers d'authentification: il existe de nombreux packages tiers qui permettent de mettre en place un système d'authentification dans Laravel, comme Laravel Breeze, Laravel JetStream et Laravel Fortify. Ces packages offrent souvent des fonctionnalités complémentaires par rapport au système d'authentification intégré de Laravel, comme la possibilité d'afficher des formulaires de connexion préconçus, la *two-factor authentication*, la vérification d'e-mail. Laravel Sanctum et Laravel Passport sont spécialisés quant à eux dans la gestion des jetons d'accès et de rafraîchissement (utile pour une API REST par exemple, ce qui est d'ailleurs le cas avec mon projet, qui repose sur Sanctum). Enfin, Laravel Passport gère l'authentification oAuth. A noter que Sanctum reste utilisable dans le cas d'un formulaire Web de connexion (explications dans la documentation de Laravel: https://laravel.com/docs/9.x/authentication#laravels-api-authentication-services).

En ce qui concerne les différences entre Sanctum et ces autres packages, voici quelques points à prendre en compte:

- Sanctum est un package Laravel qui permet de mettre en place une authentification d'API simple et sécurisée. Il est spécialement conçu pour les API et peut être utilisé pour authentifier des utilisateurs sur des applications Web et des applications mobiles.
- Sanctum utilise donc des jetons d'authentification et de rafraîchissement pour authentifier les utilisateurs, tandis que les autres packages d'authentification (excepté Passport) utilisent l'approche "session et cookie de session". Comme je l'ai dit précédemment, Laravel reste cependant compatible avec une connexion par formulaire sur un site Web.
- La documentation de Laravel recommande fortement Sanctum plutôt que d'utiliser Passport/oAuth, si cela est possible.

Je vous ai montré plus haut dans ces explications comment protéger les routes avec Sanctum. Ou, en d'autres termes, comment ne rendre accessibles certaines routes que par un utilisateur authentifié (autrement, une erreur comme quoi l'utilisateur n'est pas connecté est retournée au client): j'utilise personnellement le *middleware* `auth` avec Sanctum au niveau du fichier des routes ou dans le constructeur des contrôleurs (j'ai également déjà expliqué le concept de *middleware*). J'ai également déjà montré comment récupérer l'utilisateur actuellement connecté au sein d'une action (contrôleur ou fonction anonyme déclarée dans la définition des routes) avec l'appel `auth()->user()`.

A ce stade, je n'ai plus grand-chose à vous montrer du projet qui soit en rapport avec l'authentification et qui n'ait pas déjà été porté à votre connaissance.

Voici comment j'ai implémenté la déconnexion (c'est-à-dire la révocation du token d'authentification Sanctum):

```
        Route::post('/user/logout', function(Request $request) {
                auth()->user()->tokens()->delete();
        })->name('user_logout');
```

`tokens` est une méthode *relationship* que l'on peut appeler car le *trait* PHP `HasApiTokens` est utilisé par le modèle Eloquent `User`, qui bénéficie donc du système de tokens Sanctum:

```
class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
```

On remarque aussi que, par défaut, le modèle `User` hérite de `Authenticatable`.

6. Mise en place des tests en Laravel

Les *features tests* permettent de vérifier que votre application fonctionne correctement en exécutant des tests automatisés sur les différentes parties de votre code. Dans Laravel, les *features tests* sont écrits en utilisant le *framework* de test PHPUnit et permettent de tester les différentes routes, contrôleurs et notifications de votre application. Ils peuvent être utilisés pour tester les fonctionnalités de votre application telles que l'envoi de formulaires, la création de comptes d'utilisateur, la mise à jour de données en base de données, etc. En écrivant des *features tests* de manière cohérente et en les exécutant régulièrement, vous pouvez être sûr que votre application fonctionne correctement et que les modifications apportées ne causent pas de régressions indésirables.

Laravel ne propose pas qu'un système de *features tests*, mais aussi un système de *unit tests*. Les *unit tests* et les *features tests* sont deux types de tests différents qui ont des objectifs et des utilisations spécifiques dans le développement de logiciels.

Les *unit tests* sont utilisés pour tester des parties individuelles de votre code, telles que des fonctions ou des méthodes. Ils permettent de vérifier que chaque unité de code fonctionne

correctement et que les résultats retournés sont ceux attendus. Les *unit tests* sont généralement écrits pour des parties de code de bas niveau, telles que des fonctions de calcul ou de manipulation de données. Personnellement, je n'en ai pas écrit pour ce projet simple. Ecrire des *features tests* seulement me semblait suffisant et souhaitable.

Les *features tests*, en revanche, sont utilisés pour tester l'ensemble des fonctionnalités de votre application. Ils permettent de simuler l'utilisation de votre application par un utilisateur final et de vérifier que les différentes routes, contrôleurs et vues de votre application fonctionnent correctement ensemble. Les *features tests* sont généralement écrits pour des parties de code de haut niveau, telles que des pages Web ou des formulaires.

J'ai écrit une soixantaine de *features tests* pour mon projet. Je vais vous en montrer quelques-uns: ils concernent tous la fonctionnalité de mise à jour en base de données d'un job. En voici la liste:

— Tentative de mise à jour par un utilisateur non-authentifié.
— Tentative de mise à jour par un candidat au lieu d'une entreprise.
— Tentative de mise à jour en spécifiant un champ inattendu (c'est-à-dire non utilisé par les règles de validation du `FormRequest`), par exemple l'ID. Si `validated()`, la méthode Laravel qui ne retourne que les champs et leur valeur envoyés par le client et qui ont passé avec succès les règles de validation des `FormRequest` par exemple (si ces règles ont toutes été passées avec succès) est bel et bien utilisée, alors le test doit faire l'assertion: "une erreur de validation concernant le champ ID, qui n'était pas attendu, doit être retournée au client".
— Tentative de mise à jour en spécifiant un champ attendu (c'est-à-dire utilisé par les règles de validation du `FormRequest`) mais avec une valeur qui enfreint ces règles de validation.
— Mise à jour sans encombre.

6.1. Mise à jour sans encombre, Tentative de mise à jour en spécifiant un champ inattendu et Tentative de mise à jour en spécifiant un champ attendu dont la valeur n'est pas correcte

Voici le code qui correspond à ces tests. J'utilise un *data provider* PHPUnit. Vous pouvez constater que Laravel fournit des méthodes d'assertions, par exemple `assertDatabaseMissing`, `assertDatabaseHas`, `assertSessionHasErrors` (utilisée pour récupérer les erreurs liées à la validation des données par les `FormRequest` par exemple).

`Sanctum::actingAs($firm)` permet de se connecter en tant qu'entreprise pour les tests.

La méthode PHPUnit `setUp` crée les données pour les tests et les enregistre en base de données. La méthode `create` est une méthode non pas d'Eloquent, mais de la surcouche Laravel du SGBD directement. Une autre façon de faire serait d'utiliser les *factories*, si besoin.

Enfin, `RefreshDatabase` est un *trait* PHP qui permet de réinitialiser les données dans la base de test.

```
1 <?php
2
```

```php
namespace Tests\Feature;

use App\Models\{Job, User, Role};

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;
use Illuminate\Support\Facades\Log;
use Laravel\Sanctum\Sanctum;

class JobUpdateTest extends TestCase
{
    use RefreshDatabase;

        private array $job_update_new_data;
        private Job $job_to_update;
        private array $job_with_firm_update_new_data;

        public function setUp() : void
        {
                parent::setUp();

                Role::create([
                        'title' => 'firm'
                ]);
                Role::create([
                        'title' => 'job_applier'
                ]);

                $firm = User::create([
                        'name' => 'The Firm',
                        'email' => 'test@thegummybears.test',
                        'password' => 'azerty',
                ]);
                $firm->roles()->save(Role::findOrFail('firm'));
                Sanctum::actingAs($firm);

                $this->job_to_update = Job::create([
                        'title' => 'My Super Job',
                        'firm_id' => $firm->getKey(),
                        'presentation' => 'Its presentation',
                        'min_salary' => 45000,
                        'max_salary' => 45000,
                        'working_place' => 'full_remote',
                        'working_place_country' => 'fr',
                        'employment_contract_type' => 'cdi',
                        'contractual_working_time' => '39',
                        'collective_agreement' => 'syntec',
                        'flexible_hours' => true,
                        'working_hours_modulation_system' => true
```

```
53                ]);
54
55                $this->job_update_new_data = [
56                        'title' => 'My Giga Hyper Super Job',
57                        'min_salary' => 80000,
58                        'max_salary' => 100000,
59                        'contractual_working_time' => '35',
60                ];
61
62                $this->job_with_firm_update_new_data = [
63                        'firm_id' => 2,
64                        'title' => 'My Giga Hyper Super Job',
65                        'min_salary' => 80000,
66                        'max_salary' => 100000,
67                        'contractual_working_time' => '35',
68                ];
69        }
70
71    public function test_update_job_status() : void
72    {
73        $response = $this->put(route('jobs.update', ['job' =>
              $this->job_to_update['id']]),
              $this->job_update_new_data);
74        $response->assertStatus(200);
75    }
76
77        public function test_update_job_update_data() : void
78    {
79        $response = $this->put(route('jobs.update', ['job' =>
              $this->job_to_update['id']]),
              $this->job_update_new_data);
80        $this->assertDatabaseHas('firms_jobs',
              [...$this->job_to_update->toArray(),
              ...$this->job_update_new_data]);
81    }
82
83        public function test_update_job_with_firm_status() : void
84    {
85        $response = $this->put(route('jobs.update', ['job' =>
              $this->job_to_update['id']]),
              $this->job_with_firm_update_new_data);
86        $response->assertSessionHasErrors(['firm_id']);
87    }
88
89        public function
              test_update_job_with_firm_update_data_missing() : void
90    {
91        $response = $this->put(route('jobs.update', ['job' =>
              $this->job_to_update['id']]),
              $this->job_with_firm_update_new_data);
```

```
92          $this->assertDatabaseMissing('firms_jobs',
                [...$this->job_to_update->toArray(),
                ...$this->job_with_firm_update_new_data]);
93      }
94
95          public function
                test_update_job_with_firm_update_data_exists() : void
96      {
97          $response = $this->put(route('jobs.update', ['job' =>
                $this->job_to_update['id']]),
                $this->job_with_firm_update_new_data);
98          $this->assertDatabaseHas('firms_jobs',
                $this->job_to_update->toArray());
99      }
100
101         /**
102      * @dataProvider badDataProvider
103      */
104         public function test_bad_data(
105                 $id,
106                 $title,
107                 $firm_id,
108                 $presentation,
109                 $min_salary,
110                 $max_salary,
111                 $working_place,
112                 $working_place_country,
113                 $employment_contract_type,
114                 $contractual_working_time,
115                 $collective_agreement,
116                 $flexible_hours,
117                 $working_hours_modulation_system,
118                 $expected_result
119         )
120         {
121                 $data_to_send = [
122                         'title' => $title,
123                         'presentation' => $presentation,
124                         'min_salary' => $min_salary,
125                         'max_salary' => $max_salary,
126                         'working_place' =>  $working_place,
127                         'working_place_country' =>
                                $working_place_country,
128                         'employment_contract_type' =>
                                $employment_contract_type,
129                         'contractual_working_time' =>
                                $contractual_working_time,
130                         'collective_agreement' =>
                                $collective_agreement,
131                         'flexible_hours' => $flexible_hours,
```

```
132                     'working_hours_modulation_system' =>
                            $working_hours_modulation_system,
133             ];
134
135             if(isset($firm_id)) {
136                     $data_to_send['firm_id'] = $firm_id;
137             } elseif(isset($id)) {
138                     $data_to_send['id'] = $id;
139             }
140
141             $response = $this->put(route('jobs.update', ['job'
                    => $this->job_to_update['id']]),
                    $data_to_send);
142
143             if(isset($id)) {
144                     unset($data_to_send['id']);
145                     $this->assertDatabaseMissing('firms_jobs',
                            [
146                                 'id' => $id,
147                                 ... $data_to_send
148                     ])->assertDatabaseHas('firms_jobs', [
149                             'id' => $this->job_to_update['id'],
150                             ... $data_to_send
151                     ]);
152             } else {
153                     $response->assertSessionHasErrors($expecte
                            d_result);
154             }
155     }
156
157     public function badDataProvider() : array
158     {
159             return [
160                     [
161                             'id' => null,
162                             'title' => null,
163                             'firm_id' => 5,
164                             'presentation' => null,
165                             'min_salary' => null,
166                             'max_salary' => null,
167                             'working_place' => null,
168                             'working_place_country' => null,
169                             'employment_contract_type' => null,
170                             'contractual_working_time' => null,
171                             'collective_agreement' => null,
172                             'flexible_hours' => null,
173                             'working_hours_modulation_system'
                                    => null,
174                             'expected_result' => ['firm_id'],
175                     ],
```

```
176                         [
177                                 'id' => 999,
178                                 'title' => null,
179                                 'firm_id' => null,
180                                 'presentation' => null,
181                                 'min_salary' => null,
182                                 'max_salary' => null,
183                                 'working_place' => null,
184                                 'working_place_country' => null,
185                                 'employment_contract_type' => null,
186                                 'contractual_working_time' => null,
187                                 'collective_agreement' => null,
188                                 'flexible_hours' => null,
189                                 'working_hours_modulation_system'
                                        => null,
190                                 'expected_result' => ['id'],
191                         ]
192                 ];
193         }
194 }
```

6.2. Tentative de mise à jour par un candidat au lieu d'une entreprise.

Il s'agit de se connecter avec Sanctum en tant qu'un candidat pour faire ce test. Le *middleware* `auth` couplé à Sanctum retournera une erreur HTTP 403.

```
1  <?php
2
3  namespace Tests\Feature;
4
5  use App\Models\{Job, User, Role};
6
7  use Illuminate\Foundation\Testing\RefreshDatabase;
8  use Illuminate\Foundation\Testing\WithFaker;
9  use Tests\TestCase;
10 use Illuminate\Support\Facades\Log;
11 use Laravel\Sanctum\Sanctum;
12
13 class JobUpdateBadUserTest extends TestCase
14 {
15     use RefreshDatabase;
16
17         private array $job_update_new_data;
18         private Job $job_to_update;
19         private array $job_with_firm_update_new_data;
20
```

```
21          public function setUp() : void
22          {
23                  parent::setUp();
24
25                  Role::create([
26                          'title' => 'firm'
27                  ]);
28                  Role::create([
29                          'title' => 'job_applier'
30                  ]);
31
32                  $firm = User::create([
33                          'name' => 'The Firm',
34                          'email' => 'test@thegummybears.test',
35                          'password' => 'azerty',
36                  ]);
37                  $firm->roles()->save(Role::findOrFail('firm'));
38
39                  $applier = User::create([
40                          'name' => 'The Applier',
41                          'email' => 'test@thegummybears2.test',
42                          'password' => 'azerty',
43                  ]);
44                  $applier->roles()->save(Role::findOrFail(
                        'job_applier'));
45                  Sanctum::actingAs($applier);
46
47                  $this->job_to_update = Job::create([
48                          'title' => 'My Super Job',
49                          'firm_id' => $firm->getKey(),
50                          'presentation' => 'Its presentation',
51                          'min_salary' => 45000,
52                          'max_salary' => 45000,
53                          'working_place' => 'full_remote',
54                          'working_place_country' => 'fr',
55                          'employment_contract_type' => 'cdi',
56                          'contractual_working_time' => '39',
57                          'collective_agreement' => 'syntec',
58                          'flexible_hours' => true,
59                          'working_hours_modulation_system' => true
60                  ]);
61
62                  $this->job_update_new_data = [
63                          'title' => 'My Giga Hyper Super Job',
64                          'min_salary' => 80000,
65                          'max_salary' => 100000,
66                          'contractual_working_time' => '35',
67                  ];
68
69                  $this->job_with_firm_update_new_data = [
```

```
                        'firm_id' => 2,
                        'title' => 'My Giga Hyper Super Job',
                        'min_salary' => 80000,
                        'max_salary' => 100000,
                        'contractual_working_time' => '35',
                ];
        }

    public function test_update_job_status() : void
    {
        $response = $this->put(route('jobs.update', ['job' =>
            $this->job_to_update['id']]),
            $this->job_update_new_data);
        $response->assertStatus(403);
    }
}
```

6.3. Tentative de mise à jour par un utilisateur non-authentifié.

C'est le test le plus simple: on s'attend à ce que le *middleware* `auth` couplé à Sanctum retourne une erreur HTTP 401.

```php
<?php

namespace Tests\Feature;

use App\Models\{User, Role};

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;
use Illuminate\Support\Facades\Log;
use Laravel\Sanctum\Sanctum;

class JobStoreUnauthenticatedTest extends TestCase
{
    use RefreshDatabase;

    public function test_store_job_status() : void
    {
        $response = $this->post(route('jobs.store'), []);
        $response->assertStatus(401);
    }
}
```

6.4. Tester les notifications

Laravel permet de tester qu'une notification est envoyée. `Notification::fake()` permet de ne pas envoyer les notifications lors du test, et les fonctions d'assertions de notifications peuvent alors être utilisées sans risque.

```php
<?php

namespace Tests\Feature;

use App\Models\{User, Job, Role, JobUser};
use App\Notifications\AcceptedJobApplication;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;
use Laravel\Sanctum\Sanctum;
use Illuminate\Support\Facades\Notification;

class AcceptJobApplication extends TestCase
{
        use RefreshDatabase;

        private User $applier, $firm;
        private Job $job;

        public function setUp() : void
        {
                parent::setUp();

                Notification::fake();

                Role::create([
                        'title' => 'firm'
                ]);
                Role::create([
                        'title' => 'job_applier'
                ]);

                $this->applier = User::create([
                        'name' => 'Test User',
                        'email' =>
                            'testapplier@thegummybears.test',
                        'password' => 'azerty',
                ]);
                $this->applier->roles()->save(Role::findOrFail(
                    'job_applier'));

                $this->firm = User::create([
```

```
42                      'name' => 'Test User',
43                      'email' => 'testfirm@thegummybears.test',
44                      'password' => 'azerty',
45              ]);
46              $this->firm->roles()->save(Role::findOrFail('firm'
                    ));
47
48              $this->job = Job::create([
49                      'title' => 'My Super Job',
50                      'firm_id' => $this->firm->getKey(),
51                      'presentation' => 'Its presentation',
52                      'min_salary' => 45000,
53                      'max_salary' => 45000,
54                      'working_place' => 'full_remote',
55                      'working_place_country' => 'fr',
56                      'employment_contract_type' => 'cdi',
57                      'contractual_working_time' => '39',
58                      'collective_agreement' => 'syntec',
59                      'flexible_hours' => true,
60                      'working_hours_modulation_system' => true
61              ]);
62
63              JobUser::create([
64                      'job_id' => $this->job['id'],
65                      'user_id' => $this->applier['id'],
66                      'message' =>
                            'I want to apply for this job because foobar.'
67              ]);
68
69              Sanctum::actingAs($this->firm);
70      }
71
72      public function test_job_accept_status()
73      {
74              $job_application = JobUser::where([
75                      ['job_id', $this->job['id']],
76                      ['user_id', $this->applier['id']],
77              ])->firstOrFail();
78
79          $response = $this->post(route(
                'firms.accept_or_refuse_job_application',
                [
80                      'job_application' =>
                            $job_application['id'],
81              ]), [
82                      'firm_message' =>
                            'The message the firm writes, to be read by the job
                            ,
83                      'accept_or_refuse' => true,
84              ]);
```

```
85          $response->assertStatus(201);
86      }
87
88          public function test_job_accept_data()
89          {
90                  $job_application = JobUser::where([
91                          ['job_id', $this->job['id']],
92                          ['user_id', $this->applier['id']],
93                  ])->firstOrFail();
94
95          $response = $this->post(route(
                'firms.accept_or_refuse_job_application',
                [
96                          'job_application' =>
                                $job_application['id'],
97                  ]), [
98                          'firm_message' =>
                                'The message the firm writes, to be read by the job
                                ,
99                          'accept_or_refuse' => true,
100                 ]);
101         $this->assertDatabaseHas('jobs_apps_approvals', [
102                         'id' => $response->json('id'),
103                         'job_application_id' =>
                                $job_application['id'],
104                         'firm_message' =>
                                'The message the firm writes, to be read by the job
                                ,
105                         'accepted_or_refused' => true,
106                 ]);
107         }
108
109         public function test_job_accept_notification_sent()
110         {
111                 $job_application = JobUser::where([
112                         ['job_id', $this->job['id']],
113                         ['user_id', $this->applier['id']],
114                 ])->firstOrFail();
115
116         $response = $this->post(route(
                'firms.accept_or_refuse_job_application',
                [
117                         'job_application' =>
                                $job_application['id'],
118                 ]), [
119                         'firm_message' =>
                                'The message the firm writes, to be read by the job
                                ,
120                         'accept_or_refuse' => true,
121                 ]);
```

```
122
123                 Notification::assertSentTo(
124             [$this->applier], function(AcceptedJobApplication
                    $notification, $channels) use ($job_application) {
125                                 return $notification->getJobApplic
                                        ation()->id ===
                                        $job_application->id;
126                         }
127         );
128         }
129
130         /**
131      * @dataProvider badDataProvider
132      */
133         public function test_bad_data(
134                 $job_id,
135                 $firm_message,
136                 $accept_or_refuse,
137                 $expected_result
138         )
139         {
140                 $job_application = JobUser::where([
141                         ['job_id', $this->job['id']],
142                         ['user_id', $this->applier['id']],
143                 ])->firstOrFail();
144
145                 $data_to_send = [
146                         'firm_message' => $firm_message,
147                         'accept_or_refuse' => $accept_or_refuse,
148                 ];
149
150                 if(isset($job_id)) {
151                         $data_to_send['id'] = $job_id;
152                 }
153
154                 $response = $this->post(route(
                            'firms.accept_or_refuse_job_application',
                            [
155                                 'job_application' =>
                                        $job_application['id'],
156                 ]), $data_to_send);
157
158                 if(isset($job_id)) {
159                         $this->assertDatabaseHas('firms_jobs', [
160                                 'id' => $this->job['id'],
161                                 'title' => 'My Super Job',
162                                 'firm_id' => $this->firm->getKey(),
163                                 'presentation' =>
                                        'Its presentation',
164                                 'min_salary' => 45000,
```

```
165                                 'max_salary' => 45000,
166                                 'working_place' => 'full_remote',
167                                 'working_place_country' => 'fr',
168                                 'employment_contract_type' =>
                                        'cdi',
169                                 'contractual_working_time' => '39',
170                                 'collective_agreement' => 'syntec',
171                                 'flexible_hours' => true,
172                                 'working_hours_modulation_system'
                                        => true
173                         ]);
174                 } else {
175                         $response->assertSessionHasErrors($expecte
                                d_result);
176                 }
177         }
178
179         public function badDataProvider() : array
180         {
181                 return [
182                         [
183                                 'job_id' => 1,
184                                 'firm_message' =>
                                        'The message the firm writes, to be read by
                                        ,
185                                 'accept_or_refuse' => true,
186                                 'expected_result' => []
187                         ],
188                         [
189                                 'job_id' => null,
190                                 'firm_message' =>
                                        'The message the firm writes, to be read by
                                        ,
191                                 'accept_or_refuse' => 2,
192                                 'expected_result' =>
                                        ['accept_or_refuse']
193                         ]
194                 ];
195         }
196 }
```

7. Les queues, queues workers, jobs et tasks en Laravel

NB: dans cette section, le terme ”job” ne désigne plus le métier de mon projet ni la classe Eloquent homonyme, mais une fonctionnalité de Laravel (et commune à d'autres frameworks). Celle-ci est définie dans les explications suivantes.

Dans Laravel, les *queues*, ou ”files d'attente”, sont utilisées pour déporter des tâches qui prennent

du temps au risque de bloquer l'utilisateur final ou de ralentir sa navigation, ou qui ne doivent pas être exécutées immédiatement. Par exemple, si vous avez une application qui envoie des emails, vous pouvez mettre l'envoi d'emails dans une file d'attente plutôt que de le faire immédiatement lorsque l'utilisateur envoie le formulaire. Cela permet de ne pas ralentir l'application pour l'utilisateur et d'envoyer les emails de manière asynchrone. Cela vaut aussi pour les notifications envoyées par e-mail et c'est d'ailleurs ce que j'ai mis en place.

Les *workers* de *queues* sont les processus qui s'exécutent en arrière-plan et qui sont chargés de traiter les *jobs* de la file d'attente. Ils sont configurés pour s'exécuter à intervalles réguliers et vérifier s'il y a des tâches à traiter dans la file d'attente, puis les traitent dans l'ordre.

Les *jobs* sont simplement des classes PHP qui représentent les tâches à exécuter dans la file d'attente. Ils sont créés en implémentant l'interface `Illuminate\Contracts\Queue\ShouldQueue`. Cette interface indique à Laravel que le travail doit être mis dans une file d'attente plutôt que d'être exécuté immédiatement. Vous pouvez définir la logique de votre tâche dans la méthode `handle` de votre classe de travail.

En résumé, les *queues*, les *queues workers* et les *jobs* sont des outils pratiques pour déporter des tâches qui prennent du temps ou qui ne doivent pas être exécutées immédiatement, ce qui peut améliorer les performances de votre application et rendre son exécution plus efficace.

Un mot concernant un autre système dans Laravel, très similaire aux *jobs*. Dans Laravel, une tâche (*task*) est un code à exécuter périodiquement grâce à une commande de planification de tâches (*tasks scheduler*). Les tâches sont configurées dans la méthode `app/Console/Kernel.php::schedule`. Vous pouvez définir la fréquence à laquelle la tâche doit être exécutée, ainsi que la commande à exécuter pour exécuter la tâche.

Les *jobs* sont similaires aux tâches en ce qu'ils représentent une tâche à exécuter, mais ils sont gérés de manière différente. Les *jobs* sont mis dans une file d'attente et exécutés par un *queue worker*. Cela signifie qu'au lieu d'être exécutés périodiquement, les *jobs* sont exécutés lorsqu'un *queue worker* vérifie la file d'attente et trouve un travail à exécuter.

Ainsi, les *tasks* sont exécutées périodiquement grâce à une commande de planification de tâches, tandis que les *jobs* sont mis dans une file d'attente et exécutés par un *worker* de file d'attente lorsqu'un travail est disponible.

7.1. Exemples

Mon projet utilise le système de *jobs* de Laravel uniquement pour les notifications. Je n'ai donc pas créé de *job* en tant que tel. Les explications concernant les notifications ont fait l'objet d'une section plus haut.

Vous trouverez davantage d'informations au sujet des *jobs* dans la documentation de Laravel: https://laravel.com/docs/9.x/queues .

Similairement, je n'ai pas eu besoin de *tasks*: aussi je vous conseille de jeter un coup d'oeil à la documentation de Laravel prévue pour traiter cette notion, https://laravel.com/docs/9.x/scheduling .

8. Fichiers d'environnement et Fichiers de configuration en Laravel

En Laravel, les fichiers de configuration se trouvent dans le dossier *config* de l'application et définissent des options de configuration de l'application (en rapport avec la base de données dans `config/database.php`, les mails dans `config/mail.php`, ou même des options que vous avez vous-même définies). Ils sont utilisés pour personnaliser l'application en fonction de l'environnement dans lequel elle est exécutée, puisque qu'ils font souvent appel à la fonction `env` pour définir la valeur de leurs options de configuration. La fonction `env` retourne une valeur qui dépend de l'environnement d'exécution de l'application, trouvée dans un fichier d'environnement.

Les fichiers d'environnement se trouvent à la racine de l'application et leur nom se termine par `.env`. Ils sont utilisés pour définir des variables d'environnement qui sont utilisées par l'application pour personnaliser son comportement en fonction de l'environnement dans lequel elle est exécutée. On peut avoir un fichier d'environnement par contexte d'exécution (en d'autres termes: un pour une exécution en phase de développement, un autre pour la phase de tests avec `env.testing`, un autre pour la phase de production, mais on peut avoir des *scenarii* plus complexes également). L'environnement à utiliser peut être défini par la variable `APP_ENV` du fichier originel d'environnement, `.env` tout court ou, en priorité, au niveau du serveur directement.

Il est important de noter que les fichiers d'environnement ne doivent pas être versionnés dans votre référentiel de code, car ils peuvent contenir des informations sensibles ou spécifiques à chaque environnement. Vous devriez plutôt inclure ces fichiers dans votre fichier *.gitignore*.

Au lieu d'utiliser les variables d'environnement directement dans les actions (contrôleurs ou fonctions anonymes de routes) ou dans les middlewares, etc., il est recommandé de définir les valeurs de configuration dans les fichiers de configuration de Laravel et de les utiliser dans les contrôleurs en utilisant la méthode `config`. Dans la phase de production, vous pouvez utiliser la commande `php artisan config:cache` pour mettre en cache dans un seul fichier toutes les options de configurations et leur clé de configuration, pour contribuer à optimiser la vitesse de chargement du site ou de l'API. Le problème est que si la configuration est mise en cache à l'aide de cette commande, le fichier d'environnement ne seront plus chargés: les appels à la fonction `env()`, qui retourne la configuration définie dans le fichier d'environnent en cours d'utilisation, retourneront la configuration système et non celle du fichier d'environnement. C'est la raison pour laquelle, comme je le disais plus haut, "il est recommandé de définir les valeurs de configuration dans les fichiers de configuration de Laravel et de les utiliser dans les contrôleurs en utilisant la méthode `config`".

Bien entendu, l'option de configuration `APP_DEBUG` est disponible dans Laravel comme les autres *frameworks* et ne devrait être mise à `true` que dans la phase de développement (ce qui est le cas dans mon projet).

Pour en savoir plus: https://laravel.com/docs/9.x/configuration .

9. Le système de traductions de Laravel

Laravel fournit un système de traduction pour faciliter la mise en œuvre de votre application dans plusieurs langues. Laravel utilise les fichiers de traduction pour stocker les chaînes de caractères traduites dans différentes langues. Vous pouvez utiliser la fonction de traduction de Laravel dans votre code PHP ou dans vos vues Blade pour afficher des chaînes traduites. J'ai déjà mentionné Blade plus haut; il s'agit du système de *templating* fourni par défaut dans Laravel.

Pour utiliser le système de traduction de Laravel, vous devez définir la langue par défaut de votre application dans le fichier de configuration *config/app.php*:

```
    /*
    |--------------------------------------------------------------------------
    | Application Locale Configuration
    |--------------------------------------------------------------------------
    |
    | The application locale determines the default locale that will be used
    | by the translation service provider. You are free to set this value
    | to any of the locales which will be supported by the application.
    |
    */

    'locale' => 'en',
```

Ensuite, vous devez créer des fichiers de traduction pour chaque fonctionnalité nécessitant la prise en charge de la traduction. Ces fichiers de traduction sont stockés dans le répertoire *resources/lang/fr* (pour le français).

Exemple de mon fichier de traduction `resources/lang/fr/roles.php`:

```php
<?php
return [
        'job_applier' => 'Applier',
        'firm' => 'Firm'
];
```

Comme mentionné plus haut, pour utiliser ces traductions dans votre application, vous pouvez utiliser la fonction de traduction de Laravel `__()`:

```php
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
        /**
     * The "data" wrapper that should be applied.
     *
     * @var string|null
     */
    public static $wrap = 'user';

    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array|\Illuminate\Contracts\Support\Arrayable|\JsonSerializa
     */
    public function toArray($request)
    {
        return [
                        'id' => $this->getKey(),
                        'name' => $this->name,
                        'email' => $this->email,
                        'translated_roles' =>
                            $this->roles->map(function ($role) {
                                return __('roles.' . $role->title);
                        }),
                        'roles' =>  $this->roles->map(function
                            ($role) {
                                return $role->title;
                        }),
                ];
    }
}
```

Conclusion

Laravel est-il un *framework* pertinent à utiliser pour les développeurs *back-end*?

Dans cet article, nous avons exploré comment j'ai utilisé Laravel pour créer une API REST pour ma webapp CRUD. Nous avons également examiné certaines des fonctionnalités de base de Laravel, telles que l'ORM Eloquent, Sanctum pour l'authentification des utilisateurs, les

queues, les fichiers d'environnement et les fichiers de configuration, ainsi que le système de traductions.

Il est clair que Laravel est un cadre de développement web très puissant et facile à utiliser, qui offre de nombreuses fonctionnalités utiles pour la création d'applications web et d'API REST. Grâce à ses nombreux plugins - que je n'ai pas eu l'occasion de vous faire découvrir dans le cadre de ces explications - et outils natifs de développement, il permet aux développeurs de créer aisément des applications web de qualité. La documentation, réputée bien ordonnée, concrète et pratique, ainsi que la communauté de développeurs, sont également des atouts appréciables.

En fin de compte, Laravel s'est avéré être un choix judicieux pour la création de mon API REST, et je recommanderais certainement ce cadre de développement à d'autres développeurs qui cherchent à créer facilement des applications web de qualité.

Avez-vous déjà utilisé Laravel pour créer une application web ou une API REST? Si oui, quelles ont été vos expériences et quelles fonctionnalités avez-vous trouvées particulièrement utiles? *Quid* de Symfony? N'hésitez pas à écrire un commentaire.