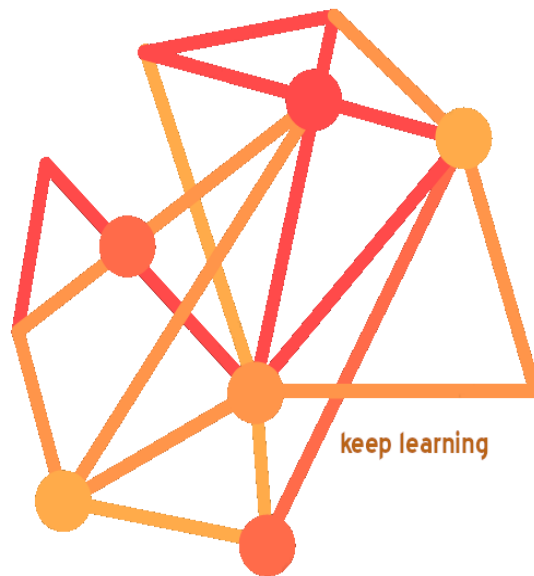


Laravel: Api REST



Juan Carlos Pérez Rodríguez

Sumario

Api REST.....	4
Crear tablas en la DDBB con migraciones / Obtener modelos desde tablas con krlove.....	4
Crear un Resource (un DTO que se devuelve automáticamente como JSON).....	8
Filtrando en la api.....	11
Swagger con laravel.....	12
Agregar autorización por token jwt.....	13
Agregar middleware de roles que los verifique del token.....	18
Anexo: subir ficheros a laravel api REST.....	20

Juan Carlos Pérez Rodríguez

Api REST

Vamos a empezar con un nuevo proyecto desde cero (aunque no sería necesario) para crear nuestra api:

```
composer create-project laravel/laravel institutoapi
```

Modificamos convenientemente nuestro fichero institutoapi/.env

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=instituto
DB_USERNAME=root
DB_PASSWORD=1q2w3e4r
```

Crear tablas en la DDBB con migraciones / Obtener modelos desde tablas con krlove

Una forma cómoda es hacer las migraciones y que cree las tablas en la DDBB:

```
php artisan make:migration create_tasks_table
```

Con lo anterior tendremos una migración (tenemos intención de crear una tabla tareas)

Editamos el fichero de migración creado:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('tasks', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
            $table->string('asunto',100)->nullable(false);
            $table->boolean('terminada')->default(false);
        });
    }
}
```

y para ejecutar la migración y que nos fabrique las tablas:

```
php artisan migrate
```

Veamos el paso contrario: Obtener las clases del modelo desde tablas de la base de datos.

Generamos las clases del modelo desde la DDBB con el módulo de krlove:

```
composer require krlove/eloquent-model-generator --dev
php artisan krlove:generate:model Task --table-name=tasks --no-timestamps
...
```

Los puntos suspensivos representan el resto de comandos generate:model para cada tabla de nuestra base de datos.

El: `--no-timestamps` es para los casos en los que nuestras tablas en la base de datos no tengan esas columnas . En caso de que no ocurra modificar a mano el fichero

Vamos a utilizar la api resources que viene con laravel. Una clase Recurso(resource) representa una clase de nuestro modelo que va a pasarse a JSON (al estilo de un DTO). También existe la posibilidad de que represente a una colección de clases del modelo que vamos a pasar a JSON.

Pero primero vamos a crear un controlador:

- crear controlador

```
php artisan make:controller ProductoRestController --resource --model=Producto
```

El parámetro: “--model” especifica la clase del modelo a la que le queremos crear un controlador. Para que nos genere los métodos típicos de un CRUD usamos “--resource”

Nos habrá creado la estructura:

index() → está pensada para gestionar el GET de todos los elementos:

GET /api/productos

store() → recibe un elemento que representa al modelo en json y lo guarda

show() → recibe peticiones del tipo:

GET /api/productos/23

siendo 23 el id del producto. Así show() muestra ese producto en json

update() → para modificar un objeto:

PUT /api/productos/23

destroy() → para destruir el objeto: DELETE /api/productos/23

```
class ProductosController extends Controller
{
    /** ...
    public function index()
    { ...
    }

    /** ...
    public function store(Request $request)
    { ...
    }

    /** ...
    public function show(Producto $producto)
    { ...
    }

    /** ...
    public function update(Request $request, Producto $producto)
    { ...
    }

    /** ...
    public function destroy(Producto $producto)
    { ...
    }
}
```

Vamos a poner un ejemplo sencillo en index():

```
class TaskController extends Controller
{
    public function index()
    {
        return response()->json([
            'foo' => 'bar',
        ]);
    }
}
```

Para generar las rutas de la api y que el método index() anterior responda a: api/tasks vamos a usar apiResource:

- ruta en: routes/api.php:

```
Route::apiResource('tasks', 'TasksController');
```

es importante que pongamos el nombre en **plural: tasks** de la clase del modelo que queremos tratar como recurso. Ya que Laravel nos crea las rutas pensando en una situación así

● **Práctica 1:** Crear el proyecto desde cero y seguir los pasos descritos: modificar apropiadamente las relaciones manytomany con los belongsto (Lo ideal sería crear las migraciones, de estas con krlove obtener las clases del modelo y hacer las modificaciones en la manytomany)

● **Práctica 2:** Crear un controller: AlumnoRestController siguiendo el patrón antes definido. Poner en: index() que devuelva: “saludo”=>”Hola soy (nombrealumno)” “Acceder mediante Rested (o equivalente) a la ruta: api/alumnos Comprobar que devuelve el mensaje

Nos vamos a poner en el caso de que hayamos creado una api con tabla Productos. Entonces:

Veamos las rutas que laravel nos ha creado con el comando: `php artisan route:list`

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/		Closure	web
	GET HEAD	api/productos	productos.index	App\Http\Controllers\ProductosController@index	api
	POST	api/productos	productos.store	App\Http\Controllers\ProductosController@store	api
	GET HEAD	api/productos/{producto}	productos.show	App\Http\Controllers\ProductosController@show	api
	PUT PATCH	api/productos/{producto}	productos.update	App\Http\Controllers\ProductosController@update	api
	DELETE	api/productos/{producto}	productos.destroy	App\Http\Controllers\ProductosController@destroy	api
	GET HEAD	api/user		Closure	api,auth:api

vemos que las rutas empiezan todas con: /api Fijémonos también en las rutas que genera para cuando el usuario envía request para un producto concreto:

api/productos/{producto}

laravel tomó la palabra que nosotros le dimos en plural y estableció un parámetro de path con la palabra en singular: {producto} Siendo éste el nombre correcto para nuestro objeto de la clase modelo. Es por lo anterior por lo que debemos poner el nombre en plural al crear la ruta

Podemos siempre agregar otras rutas mediante: Route. Ejemplo: api/tasks/miaccion:

```
Route::prefix(")->group(function () {  
    Route::get('/tasks/miaccion', [TaskController::class,"mimetodo"]);  
  
    // ... (rutas adicionales)  
});
```

● **Práctica 3:** Crear para la api todos los puntos en api.php de rutas de resource para nuestros objetos: Alumno, Matricula, Asignatura. Busca información de como excluir el borrado de ApiResource("alumnos") y ponlo luego como una linea adicional (la forma que hemos hecho siempre, mediante: Route::delete...) Después toma captura de pantalla de la salida de php artisan route:list

Crear un Resource (un DTO que se devuelve automáticamente como JSON)

- crear recurso para la clase del modelo(controla que mostramos para cada entidad del modelo)

```
php artisan make:resource TaskDTO
```

No es obligatorio agregar la palabra Resource al nombre del recurso

El anterior comando habrá creado el fichero: app/Http/Resources/TaskDTO.php

- Usando el Resource en el return del controller (TaskController):

```
public function index()  
{  
    return TaskDTO::collection(Task::all());  
}
```


Observar que nos estamos apoyando en eloquent: `Task::all()` que nos devuelve todas las tareas de la base de datos. Luego ejecutamos un return del resource: `TaskDTO` envolviendo esas tareas: `TaskDTO::collection(Task::all())`

Con lo anterior si ejecutamos: **api/tasks** en postman o rested veremos la lista de tareas de la base de datos

Podemos hacer el Resource personalizado (como un DTO)

En la function `toArray()` escribimos la información para mostrar del recurso. Vamos a ver un ejemplo para la tabla productos:

```
public function toArray($request)
{
    //el texto keyMostrada se pone únicamente para observar que
    //lo que pongamos ahí lo devolverá en el json.
    //En un resource la palabra: $this hace referencia a la clase del modelo
    //así nosotros debemos poner los nombres de las propiedades del modelo:
    //$this->idproducto significa que queremos la propiedad: idproducto del modelo: Producto
    return [
        'idkeyMostrada' => $this->idproducto,
        'nombreKeyMostrada' => $this->nombre,
        'precioKeyMostrada' => $this->precio,
        'stockKeyMostrada' => $this->stock,
    ];
}
```

No hemos visto la parte del controlador para crear producto, borrar, editar.
Para crear podemos tener algo como:

```
public function store(Request $request)
{
    $producto = Producto::create([
        'nombre' => $request->nombre,
        'precio' => $request->precio,
    ]);

    return new ProductoDTO($producto);
}
```

En la devolución del ProductoResource podemos ver el idproducto creado automáticamente

Para borrar:

```
public function destroy(Producto $producto)
{
    $producto->delete();
    return response()->json(null, 204);
}
```

Observar: Producto \$producto Laravel nos hace una inyección interesante. Ya que el usuario envía peticiones con id: DELETE /api/productos/21 Y de esa forma quiere borrar el producto con id 21. Pues bien, laravel transforma ese id en el objeto Producto que corresponde

También vemos que podemos dar un código 204 a nuestra respuesta para informar del éxito de la petición: response()->json(null, 204);

Ahora el update():

```
public function update(Request $request, Producto $producto)
{
    $producto->update($request->only(['nombre', 'precio']));
    return new ProductoResource($producto);
}
```

Vemos que nos vuelve a inyectar \$producto a una petición por id del usuario. Hemos elegido mostrar una actualización únicamente para dos parámetros enviados en la request: nombre, precio

- Práctica 4:** Crear el CRUD y que funcione para la api instituto con Asignatura, Alumno, Matricula. Para la parte de como se hace un update en Matricula no complicarse con la tabla intermedia asignatura_matricula. Borrar todos los registros que haya e insertar los que correspondan con la lista de asignaturas que se reciba en el PUT
Nota: recordar que no es obligatorio llenar el array de asignaturas de la matricula con toda la información. Se puede únicamente enviar el id y que ya la api haga lo que corresponda

Filtrando en la api

Como `index()` debe devolver una colección. Un ejemplo del código para una consulta en la que el usuario quiere los productos filtrados por nombre: **GET /api/productos?nombre=pan**

```
public function index(Request $request)
{
    $parametroKey = 'nombre';
    $parametroValue = $request->input($parametroKey);

    ...
}
```

Observamos que hemos inyectado: `Request $request` para acceder a los parámetros de la solicitud.

Una vez realizado lo anterior ya podemos usar postman, rested o la utilidad que prefiramos para testear y ver el json de todos los productos de nuestra tabla productos (recordar que el ejemplo anterior estaba pensado para recibir un parámetro nombre) Un ejemplo sería:

`http://localhost:8001/api/productos?nombre=Television`

- **Práctica 5:** Agregar a la api que podamos filtrar los alumnos por nombre: `api/alumnos?nombre` y también por apellidos: `api/alumnos?apellidos` En ambos casos debe soportar que lo que se envíe sea únicamente un trozo. Así por ejemplo si se pone: `api/alumnos?nombre=to` hará match con: `nombre=roberto`, `nombre=tolomeo`, ...

Swagger con laravel

Instalamos el paquete:

```
composer require "darkaonline/l5-swagger"
```

Luego generamos el fichero de configuración y establecemos como un provider:

```
php artisan vendor:publish --provider "L5Swagger\L5SwaggerServiceProvider"
```

Hay que poner anotaciones en la raíz y en cada endpoint a documentar. En la Dirección raíz de nuestros controller(por ejemplo anotando justo encima de: class Controller() ya que todos nuestros controller heredan de ahí) ponemos @OA\Info


```
/**
 * @OA\Info(
 *   title="Instituto api",
 *   version="1.0.0",
 *   description="Esta es la documentación de la API generada automáticamente con Swagger",
 *   @OA\Contact(
 *     email="soporte@example.com"
 *   )
 * )
 */
```

Luego En los endpoint que queramos documentar debemos anotarlos. Por ejemplo en AlumnoRestController::index() lo anotamos para que las peticiones: GET /api/alumnos las incluya:

```
/**
 * @OA\Get(
 *   path="/api/alumnos",
 *   summary="Obtener lista de alumnos",
 *   description="Retorna una lista de alumnos",
 *   tags={"Alumnos"},
 *   @OA\Response(
 *     response=200,
 *     description="Lista de alumnos"
 *   )
 * )
 */
public function index(){ ... }
```

Finalmente generamos la documentación y accedemos a ella en la ruta: </api/documentation>

```
php artisan l5-swagger:generate
```

 **Práctica 5.1:** Documentar la api. Investiga un poco las diferentes opciones que puedes usar y explica que hace cada opción que has puesto

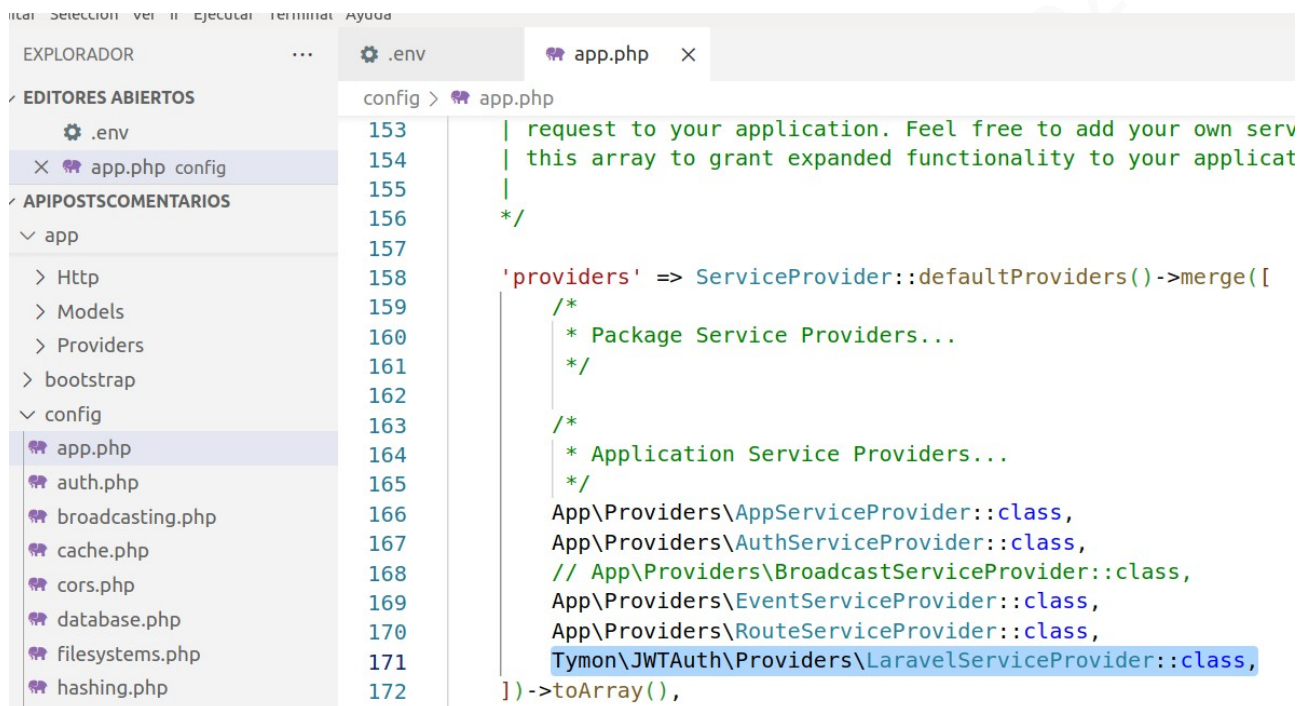
Agregar autorización por token jwt

Primero necesitamos instalar la librería:

```
composer require tymon/jwt-auth
```

Hay que agregar una línea en: **config/app.php**, en la parte de 'providers':

```
Tymon\JWTAuth\Providers\LaravelServiceProvider::class,
```



```
153 | request to your application. Feel free to add your own serv
154 | this array to grant expanded functionality to your applicat
155 |
156 |
157 |
158 |
159 |
160 |
161 |
162 |
163 |
164 |
165 |
166 |
167 |
168 |
169 |
170 |
171 |
172 |
173 |
174 |
175 |
176 |
177 |
178 |
179 |
180 |
181 |
182 |
183 |
184 |
185 |
186 |
187 |
188 |
189 |
190 |
191 |
192 |
193 |
194 |
195 |
196 |
197 |
198 |
199 |
200 |
201 |
202 |
203 |
204 |
205 |
206 |
207 |
208 |
209 |
210 |
211 |
212 |
213 |
214 |
215 |
216 |
217 |
218 |
219 |
220 |
221 |
222 |
223 |
224 |
225 |
226 |
227 |
228 |
229 |
230 |
231 |
232 |
233 |
234 |
235 |
236 |
237 |
238 |
239 |
240 |
241 |
242 |
243 |
244 |
245 |
246 |
247 |
248 |
249 |
250 |
251 |
252 |
253 |
254 |
255 |
256 |
257 |
258 |
259 |
260 |
261 |
262 |
263 |
264 |
265 |
266 |
267 |
268 |
269 |
270 |
271 |
272 |
273 |
274 |
275 |
276 |
277 |
278 |
279 |
280 |
281 |
282 |
283 |
284 |
285 |
286 |
287 |
288 |
289 |
290 |
291 |
292 |
293 |
294 |
295 |
296 |
297 |
298 |
299 |
300 |
301 |
302 |
303 |
304 |
305 |
306 |
307 |
308 |
309 |
310 |
311 |
312 |
313 |
314 |
315 |
316 |
317 |
318 |
319 |
320 |
321 |
322 |
323 |
324 |
325 |
326 |
327 |
328 |
329 |
330 |
331 |
332 |
333 |
334 |
335 |
336 |
337 |
338 |
339 |
340 |
341 |
342 |
343 |
344 |
345 |
346 |
347 |
348 |
349 |
350 |
351 |
352 |
353 |
354 |
355 |
356 |
357 |
358 |
359 |
360 |
361 |
362 |
363 |
364 |
365 |
366 |
367 |
368 |
369 |
370 |
371 |
372 |
373 |
374 |
375 |
376 |
377 |
378 |
379 |
380 |
381 |
382 |
383 |
384 |
385 |
386 |
387 |
388 |
389 |
390 |
391 |
392 |
393 |
394 |
395 |
396 |
397 |
398 |
399 |
400 |
401 |
402 |
403 |
404 |
405 |
406 |
407 |
408 |
409 |
410 |
411 |
412 |
413 |
414 |
415 |
416 |
417 |
418 |
419 |
420 |
421 |
422 |
423 |
424 |
425 |
426 |
427 |
428 |
429 |
430 |
431 |
432 |
433 |
434 |
435 |
436 |
437 |
438 |
439 |
440 |
441 |
442 |
443 |
444 |
445 |
446 |
447 |
448 |
449 |
450 |
451 |
452 |
453 |
454 |
455 |
456 |
457 |
458 |
459 |
460 |
461 |
462 |
463 |
464 |
465 |
466 |
467 |
468 |
469 |
470 |
471 |
472 |
473 |
474 |
475 |
476 |
477 |
478 |
479 |
480 |
481 |
482 |
483 |
484 |
485 |
486 |
487 |
488 |
489 |
490 |
491 |
492 |
493 |
494 |
495 |
496 |
497 |
498 |
499 |
500 |
501 |
502 |
503 |
504 |
505 |
506 |
507 |
508 |
509 |
510 |
511 |
512 |
513 |
514 |
515 |
516 |
517 |
518 |
519 |
520 |
521 |
522 |
523 |
524 |
525 |
526 |
527 |
528 |
529 |
530 |
531 |
532 |
533 |
534 |
535 |
536 |
537 |
538 |
539 |
540 |
541 |
542 |
543 |
544 |
545 |
546 |
547 |
548 |
549 |
550 |
551 |
552 |
553 |
554 |
555 |
556 |
557 |
558 |
559 |
560 |
561 |
562 |
563 |
564 |
565 |
566 |
567 |
568 |
569 |
570 |
571 |
572 |
573 |
574 |
575 |
576 |
577 |
578 |
579 |
580 |
581 |
582 |
583 |
584 |
585 |
586 |
587 |
588 |
589 |
590 |
591 |
592 |
593 |
594 |
595 |
596 |
597 |
598 |
599 |
600 |
601 |
602 |
603 |
604 |
605 |
606 |
607 |
608 |
609 |
610 |
611 |
612 |
613 |
614 |
615 |
616 |
617 |
618 |
619 |
620 |
621 |
622 |
623 |
624 |
625 |
626 |
627 |
628 |
629 |
630 |
631 |
632 |
633 |
634 |
635 |
636 |
637 |
638 |
639 |
640 |
641 |
642 |
643 |
644 |
645 |
646 |
647 |
648 |
649 |
650 |
651 |
652 |
653 |
654 |
655 |
656 |
657 |
658 |
659 |
660 |
661 |
662 |
663 |
664 |
665 |
666 |
667 |
668 |
669 |
670 |
671 |
672 |
673 |
674 |
675 |
676 |
677 |
678 |
679 |
680 |
681 |
682 |
683 |
684 |
685 |
686 |
687 |
688 |
689 |
690 |
691 |
692 |
693 |
694 |
695 |
696 |
697 |
698 |
699 |
700 |
701 |
702 |
703 |
704 |
705 |
706 |
707 |
708 |
709 |
710 |
711 |
712 |
713 |
714 |
715 |
716 |
717 |
718 |
719 |
720 |
721 |
722 |
723 |
724 |
725 |
726 |
727 |
728 |
729 |
730 |
731 |
732 |
733 |
734 |
735 |
736 |
737 |
738 |
739 |
740 |
741 |
742 |
743 |
744 |
745 |
746 |
747 |
748 |
749 |
750 |
751 |
752 |
753 |
754 |
755 |
756 |
757 |
758 |
759 |
760 |
761 |
762 |
763 |
764 |
765 |
766 |
767 |
768 |
769 |
770 |
771 |
772 |
773 |
774 |
775 |
776 |
777 |
778 |
779 |
780 |
781 |
782 |
783 |
784 |
785 |
786 |
787 |
788 |
789 |
790 |
791 |
792 |
793 |
794 |
795 |
796 |
797 |
798 |
799 |
800 |
801 |
802 |
803 |
804 |
805 |
806 |
807 |
808 |
809 |
810 |
811 |
812 |
813 |
814 |
815 |
816 |
817 |
818 |
819 |
820 |
821 |
822 |
823 |
824 |
825 |
826 |
827 |
828 |
829 |
830 |
831 |
832 |
833 |
834 |
835 |
836 |
837 |
838 |
839 |
840 |
841 |
842 |
843 |
844 |
845 |
846 |
847 |
848 |
849 |
850 |
851 |
852 |
853 |
854 |
855 |
856 |
857 |
858 |
859 |
860 |
861 |
862 |
863 |
864 |
865 |
866 |
867 |
868 |
869 |
870 |
871 |
872 |
873 |
874 |
875 |
876 |
877 |
878 |
879 |
880 |
881 |
882 |
883 |
884 |
885 |
886 |
887 |
888 |
889 |
890 |
891 |
892 |
893 |
894 |
895 |
896 |
897 |
898 |
899 |
900 |
901 |
902 |
903 |
904 |
905 |
906 |
907 |
908 |
909 |
910 |
911 |
912 |
913 |
914 |
915 |
916 |
917 |
918 |
919 |
920 |
921 |
922 |
923 |
924 |
925 |
926 |
927 |
928 |
929 |
930 |
931 |
932 |
933 |
934 |
935 |
936 |
937 |
938 |
939 |
940 |
941 |
942 |
943 |
944 |
945 |
946 |
947 |
948 |
949 |
950 |
951 |
952 |
953 |
954 |
955 |
956 |
957 |
958 |
959 |
960 |
961 |
962 |
963 |
964 |
965 |
966 |
967 |
968 |
969 |
970 |
971 |
972 |
973 |
974 |
975 |
976 |
977 |
978 |
979 |
980 |
981 |
982 |
983 |
984 |
985 |
986 |
987 |
988 |
989 |
990 |
991 |
992 |
993 |
994 |
995 |
996 |
997 |
998 |
999 |
1000 |
```

Una vez instalada creamos el fichero: **config/jwt.php** mediante el siguiente comando:

```
php artisan vendor:publish --provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"
```

El servidor precisa una clave secreta para crear los token. La generamos mediante:

```
php artisan jwt:secret
```

Habr  creado en el fichero .env de la ra z de la carpeta la clave secreta

● **Pr ctica 6:** Buscar la l nea que incluya la palabra JWT en el fichero .env   aparece la clave ?   c mo se llama el nombre del campo donde se guarda la clave ?

Ahora en: **config/auth.php** tenemos que modificar en la parte de **guards**:

```
'api' => [
    'driver' => 'jwt',
    'provider' => 'users',
],

'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

El siguiente paso es modificar el código de nuestra clase usuario para que haga uso de jwt. Le debemos agregar la interfaz: **JWTSubject** Por ejemplo si la clase fuera Usuario pondríamos:

```
use Tymon\JWTAuth\Contracts\JWTSubject;
use Illuminate\Foundation\Auth\User as Authenticatable;
class User extends Authenticatable implements JWTSubject
{
    public function getJWTIdentifier()
    {
        return $this->getKey();
    }

    public function getJWTCustomClaims()
    {
        //lo que pongamos en este array se agrega al token
        return [
            'rol' => 'user',
            'name' => $this->name
        ];
    }
}
```

Observar que la clase no está completa Únicamente mostramos el código nuevo.

getJWTIdentifier() Obtiene el identificador para Subject que irá en el token. **getJWTCustomClaims()** nos permite agregar claims propias al token. **No es obligatorio**. Se puede agregar por ejemplo los roles

Tenemos que establecer con controlador para el login() y para el register() en la api Creamos un controlador **AuthApiController**:

```
php artisan make:controller AuthApiController
```

Editamos el fichero creado:

```
use Illuminate\Http\Request;
use App\Models\User;
use Illuminate\Support\Facades\Hash;
use Tymon\JWTAuth\Facades\JWTAuth;

class AuthApiController extends Controller
{
    public function register(Request $request)
    {
        $user = User::create([
            'name' => $request->name,
            'email' => $request->email,
            'password' => Hash::make($request->password),
        ]);
        return auth('api')->login($user);
    }

    public function login(Request $request)
    {
        $nom = $request->input('name');
        $pass = $request->input('password');
        $user = User::where('name', '=', $nom)
            ->first();

        if( isset($user) ){
            $usuarioname = $user['name'];
            $usuariohashpass = $user['password'];
            if ( Hash::check($pass, $usuariohashpass) ) {
                $token = JWTAuth::fromUser($user);
                return $token;
            } else {
                return response()
                    ->json(['error' => 'Unauthorized', $nom => $pass], 401);
            }
        } else {
            return response()
                ->json(['error' => 'User not found', $nom => $pass], 401);
        }
    }
}
```

Como se puede ver el código es casi todo personalizable y puede diferir bastante del ejemplo dado. Se ha elegido mostrar un caso de una respuesta json que incorpora el token. Lo relevante del código anterior es:

`$token = auth('api')->login($user);` Tenemos varias `auth()` ahora, así que hay que especificar la de la api: `auth('api')` Observar que como pusimos el driver `jwt` en el fichero de configuración conseguimos que la llamada a `auth('api')` nos devuelva el token `jwt`

`response()->json(['error' => 'Unauthorized'], 401)` para devolver una respuesta de prohibido

Por último especificamos las rutas en: `routes/api.php`

```
Route::post('register', [AuthApiController::class, 'register']);
Route::post('login', [AuthApiController::class, 'login']);
```

Ya podemos usar autenticación.

No olvidar que para proteger debemos especificar el middleware en las rutas `apiResource`. Así por ejemplo: `Route::apiResource('alumnos', AlumnoRestController::class)->middleware('auth:api');`

También es posible en el constructor del controlador. Así `ProductosController` queda:

```
class ProductosController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth:api')->except(['index', 'show']);
    }
}
```

Vemos que tenemos que siempre informar de que usamos `auth:api` Con lo anterior se requiere autenticación para producto salvo listar todos o mostrar uno

Podemos usar `user()` como habitualmente así por ejemplo el siguiente código prohíbe editar un libro del que no sea propietario el usuario (código incompleto sin cierre)

```
public function update(Request $request, Book $book)
{
    if ($request->user()->id !== $book->user_id) {
        return response()->json(['error' => 'You can only edit your own books.'], 403);
    }
}
```

El proceso para acceder ahora sería: <http://localhost:8001/api/login> y poner nombre/pass para obtener token

The screenshot shows the Postman interface. On the left, the 'Collections' tab is active, showing a message: 'No collected requests. Add by pressing "plus" in the top right of the request panel.' The main area displays a 'Request' tab for a POST method to 'http://localhost:8001/api/login'. The 'Request body' is set to JSON with two parameters: 'nombre' with value 'lui' and 'password' with value 'lui'. Below the request, the 'Response' tab shows a '200 OK' status and a JSON body containing an 'access_token'.

Ahora solicitamos los productos mediante el token obtenido en la ruta: <http://localhost:8001/api/productos> Nota: Escribir en el campo Header: Authorization

Bearer token

donde token es el token obtenido al hacer login

</> RESTED

The screenshot shows the Postman interface. On the left, the 'Collections' tab is active, showing a collection named 'Collection' with two requests: a GET request to 'http://localhost:8001/api/productos' and a POST request to 'http://localhost:8001/api/login'. The main area displays a 'Request' tab for a GET method to 'http://localhost:8001/api/productos'. The 'Headers' section includes 'Authorization' with a Bearer token and 'Content-Type' set to 'application/json'. Below the request, the 'Response' tab shows a '200 OK' status and a JSON body containing an 'id' field.

Práctica 7: Poner autenticación a nuestra api Se puede ver libremente las asignaturas. Pero necesario autenticación para alumnos y matriculas.

Agregando middleware de roles que los verifique del token

En la parte de: “Creando Middleware” de este pdf ya comentamos como se creaba un middleware.

Recordemos que se ejecuta:

```
php artisan make:middleware RolAdmin
```

y nos crea el fichero. Supongamos que lo rellenamos con (fichero App/Http/Middleware/RolAdmin.php)

```
use JWTAuth;
class RolAdmin
{
    public function handle($request, Closure $next)
    {
        $token = JWTAuth::parseToken();
        JWTAuth::getPayload();
        $resp = JWTAuth::getPayload()->get('rol');

        if( $resp == 'admin'){
            return $next($request);
        }else{
            return response()->json(['mensaje' => 'rol no autorizado'], 401);
        }
    }
}
```

Vemos que ya está preparado para tomar el token de la cabecera mediante: JWTAuth::parseToken() es importante poner el comando use apropiado: use JWTAuth

Observar que mediante el comando: getPayload() tomamos los claims y obtenemos el que queremos.

Recordar agregar en **Kernel.php** la declaración del middleware para usar la etiqueta en rutas

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,

    'roladmin' => \App\Http\Middleware\RolAdmin::class,
];
```

Y finalmente agregarlo a routes/api.php o en constructor del controlador (vemos línea de api.php)

```
Route::apiResource('productos', 'ProductosController')->middleware('roladmin');
```

Para terminar, una descripción bastante completa de lo que podemos hacer con JWTAuth:

<https://cubettech.com/resources/blog/api-authentication-using-jwt-in-laravel-5/>

- **Práctica 8:** Agregar el middleware roladmin. Ahora si se tiene ese rol se pueden hacer todas las acciones de la api libremente. Si se está autenticado sin ser admin se ven todas las asignaturas (igual que sin autenticar) y todas las demás acciones de lectura (matrículas, alumnos) pero ninguna de las otras acciones (delete,...) Buscar información de como excluir una acción de apiResource (las acciones get) y proteger con roladmin ese apiResource “recortado” generar las rutas adicionales que corresponda para hacer todo lo solicitado

Anexo: subir ficheros a laravel api REST

Si vamos a usar el formato binario para la subida (qué es óptimo) No es diferente de como hemos hecho subidas de ficheros con laravel. Creamos un controller (por ejemplo ImagesController) y generamos el método que gestiona la petición (en el ejemplo será upload()). Finalmente llamamos a ese método desde el fichero: **api.php**:

```
//ruta /api/upload  
Route::post('/upload', [ImagesController::class, 'upload']);
```

Fichero ImagesController:

```
class ImagesController extends Controller  
{  
    public function upload(Request $request)  
    {  
        //suponemos que el atributo al subir se ha llamado: file  
        if ($request->hasFile('file')) {  
            $fichero = $request->file('file');  
            $nombre = $fichero->getClientOriginalName();  
            $path = $fichero->storeAs($nombre);  
  
            return response()->json([  
                'message' => 'ok, fichero subido',  
                'path' => $path  
            ], 200);  
        }  
  
        return response()->json([  
            'message' => 'No se encontró ninguna imagen.',  
        ], 400);  
    }  
    ...  
}
```

Vemos un ejemplo de subir el fichero en formato original con React:

```
function UploadFichero() {
  const [mensajes, setmensajes] = useState("");

  async function subirfichero(ev: FormEvent<HTMLFormElement>){
    ev.preventDefault();
    let formulario = ev.currentTarget;
    let file = formulario.inputfichero.files[0];

    if (file) {
      const formData = new FormData();
      formData.append("file", file);
      try {
        let response = await axios.post('http://localhost:8000/api/upload',
          formData, {
            headers: { 'Content-Type': 'multipart/form-data' }
          }
        );
        let respuesta = "";

        if(response.data){ respuesta = JSON.stringify(response.data); }

        setmensajes(respuesta);
      } catch (error) { console.log("error dice: "+error); }
    }
  }

  return (
    <div >
      <form onSubmit={subirfichero}>
        <label htmlFor="file" >
          elegir fichero
        </label>
        <input id="inputfichero" type="file" />
        <button type="submit">Subir</button>
      </form>

      mensajes: {mensajes}
    </div>
  );
}
```