Laravel: Api REST

Juan Carlos Pérez Rodríguez

Sumario

Api REST

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

```
composer create-project laravel/laravel apicarrito
```

Modificamos convenientemente nuestro fichero apicarrito/.env

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=tiendadb
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

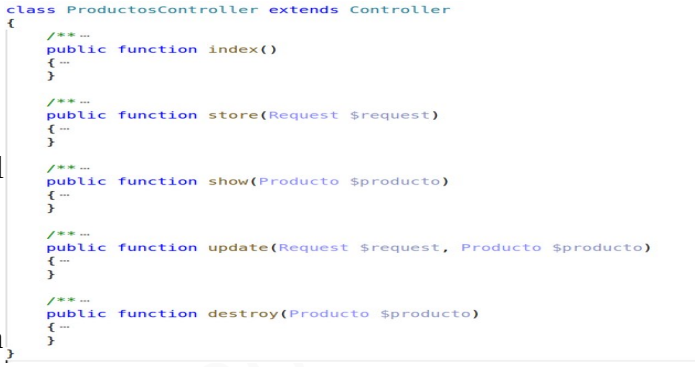
Vamos a utilizar la api resources que viene con laravel. Una clase Recurso(resource) representa una clase de nuestro modelo que va a pasarse a JSON (al estilo de un DTO). También existe la posibilidad de que represente a una colección de clases del modelo que vamos a pasar a JSON.

Pero primero vamos a crear un controlador:

- crear controlador

```
php artisan make:controller TasksController -m Task --api
```

El parámetro: “-m” especifica la clase del modelo a la que le queremos crear un controlador. Para que nos genere los métodos correctos para la api especificamos: “--api”



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

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

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)
});
```

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
```

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.

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 TaskResource
```

No es obligatorio agregar la palabra Resource al nombre del recurso

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

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

```
public function index()
{
        return TaskResource::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: TaskResource envolviendo esas tareas: TaskResource::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 ProductoResource($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

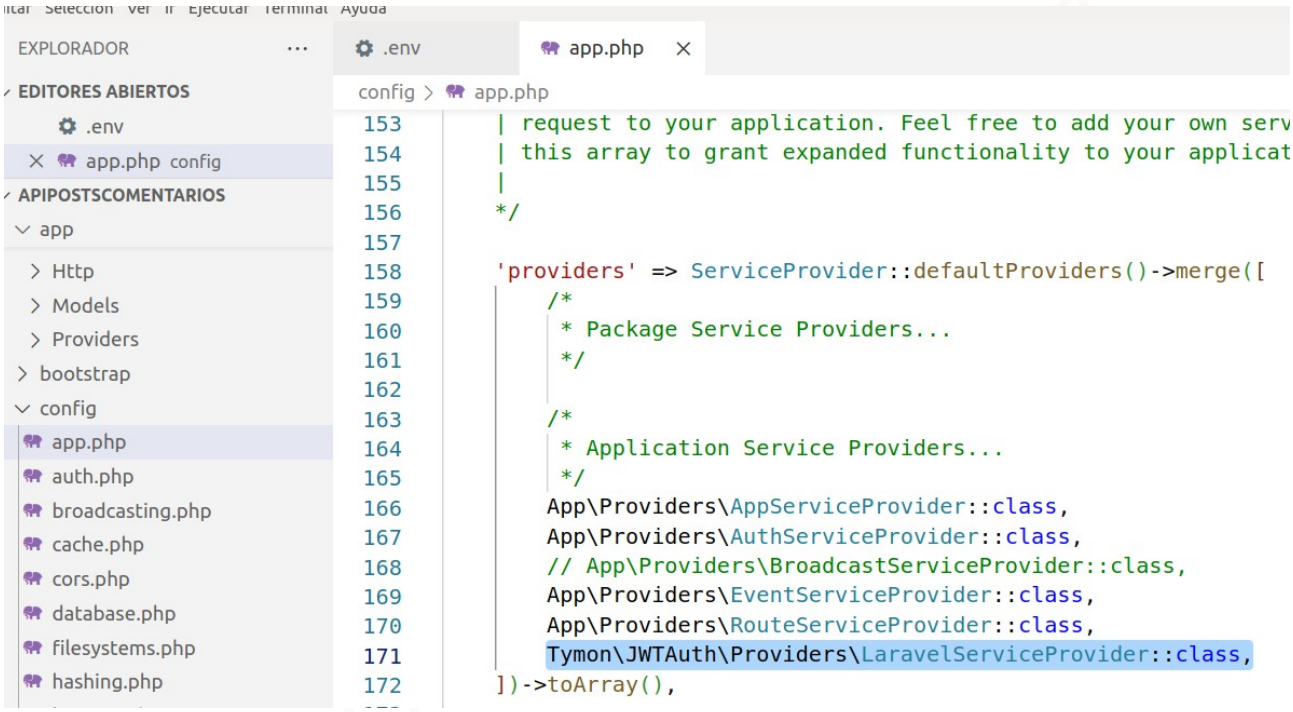
Agregar autorización por token jwt

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,
```



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
```

Habrá creado en el fichero .env de la raíz de la carpeta la clave secreta

Práctica 39: Buscar la línea que incluya la palabra JWT en el fichero .env ¿ aparece la clave ? ¿ cómo 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 tokey
                return [
                        'rol' => 'usuario',
                        '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 AuthController:

```
php artisan make:controller AuthController
```

Editamos el fichero creado:

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

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

Ya podemos usar autenticación.

No olvidar que para proteger debemos especificar el middleware en el constructor del controlador. Ahora ProductosController nos queda:

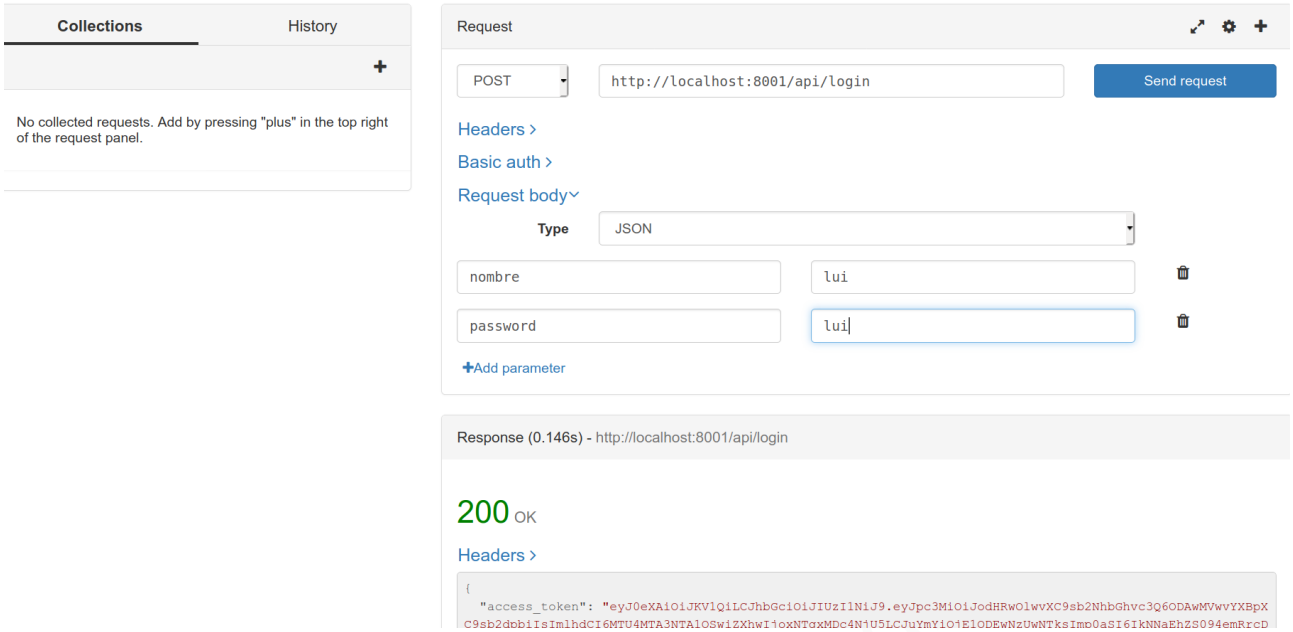
```
class ProductosController extends Controller
{
	public function __construct()
	{
		$this->middleware('auth:api')->except(['index','show']);
	}
```

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 prohibe editar un libro del que no sea propietario el usuario (código incompleto sin cierre)

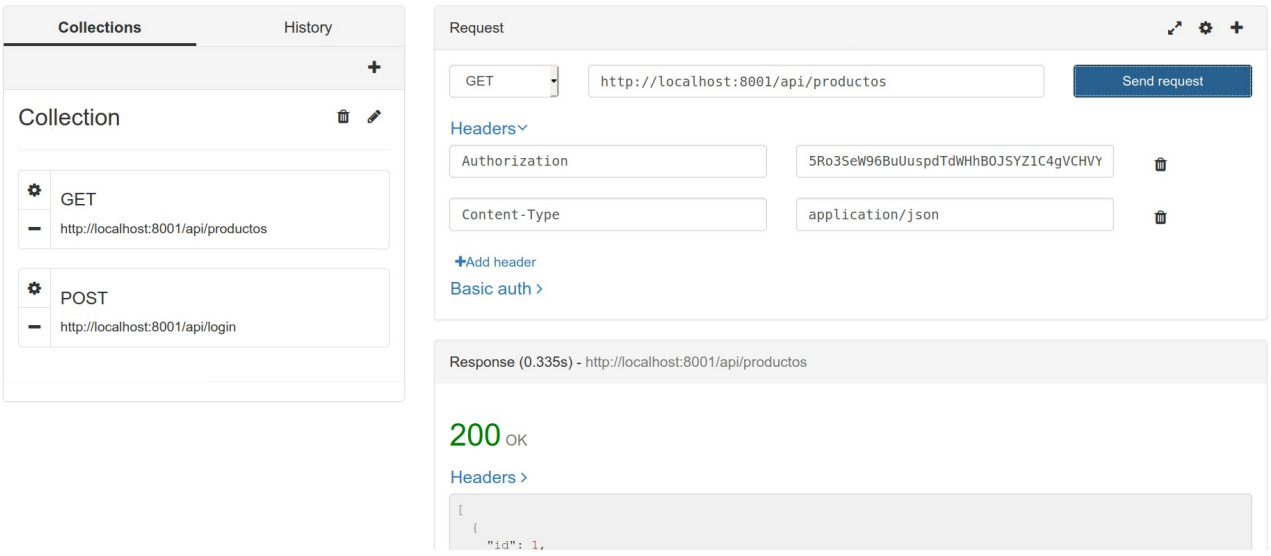
```
public function update(Request $request, Book $book)
{
	if ($request->user()->id !== $book->user_id) {
		return response()->json(['error' => 'You can only edit your own books.'], 403);
	}
```

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



Ahora solicitamos los productos mediante el token obtenido en la ruta: http://localhost:8001/api/productos

RESTED



Práctica 40: Poner autenticación a nuestra api Se puede ver libremente los productos y los comentarios. Pero necesario autenticación para carrito y usuario. La información de usuario sólo la propia y editar únicamente carrito y usuario propio

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/

Anexos

Autenticación

Con carácter general hay que tener en cuenta que las clases del modelo tienen, por defecto en laravel, campos timestamp: created_at, updated_at, etc que informan de la fecha de creación del registro y demás. Si eso no tiene coherencia con las tablas que tenemos en nuestra base de datos vamos a tener problemas. Podemos desactivar esos campos informando en la clase del modelo que no queremos que guarde en base de datos los timestamps (por ejemplo si vamos a usar una tabla usuario que teníamos de antes para la autenticación y no contempla timestamps tendremos fallos si no lo desactivamos, lo mismo ocurre con rol y usuario_rol)

```
use Illuminate\Foundation\Auth\User as Authenticatable;
class Usuario extends Authenticatable
{
        public $timestamps = false;
```

En el caso del ejemplo estamos diciendo que la tabla-clasemodelo usuario no va a registrar los timestamps

Observar que hemos hecho que extienda de Authenticatable (alias de User) esto es importante porque esta clase va a reemplazar la clase User que por defecto usa laravel en la autenticación. Si hereda de User mantiene todo el comportamiento de User y será una clase válida para el sistema de autenticación de laravel

Como hemos dicho la clase Usuario va a reemplazar a la clase User, así que tenemos que modificar unas cuantas cosas para que así sea. Sobre todo en el fichero: config/auth.php:

Para que sea más fácil identificar los cambios en la tabla de la base de datos se ha antepuesto: ddbb Así el nombre de la tabla que gestiona los usuarios es:

ddbbusuario(idusuario, ddbbnombre, ddbbclave **)**

donde idusuario es primary-key autoincremental. ddbbnombre es varchar(45) y ddbbclave es varchar(145) (recordar que los hash pueden ser grandes)

La clase modelo para esa tabla la hemos generado con:

```
php artisan krlove:generate:model ModelUsuario --table-name=ddbbusuario --no-timestamps
```

Así que la clase que queremos que sea la que gestione los usuarios se llama: ModelUsuario

(observar que hemos dejado comentado lo que estaba antes y ya no va a estar -verde- y ponemos lo que debemos establecer en azul)

Se ha puesto el nombre: **providermodelusuario** como una etiqueta para que se observe que lo único relevante es que en este fichero siempre referenciemos a ese nombre de etiqueta

```
'defaults' => [
'guard' => 'web',
'passwords' => 'providermodelusuario',
/*
'passwords' => 'users',
*/
],
```

```
'guards' => [
/*
'web' => [
'driver' => 'session',
'provider' => 'users',
],
*/
'web' => [
'driver' => 'session',
'provider' => 'providermodelusuario',
],
```

```
'providers' => [
'users' => [
'driver' => 'eloquent',
'model' => App\User::class,
```

```
],

'providermodelusuario' => [
'driver' => 'eloquent',
'model' => App\ModelUsuario::class,
],
],
```

```
'passwords' => [
'users' => [
'provider' => 'users',
'table' => 'password_resets',
'expire' => 60,
'throttle' => 60,
],
'providermodelusuario' => [
'provider' => 'providermodelusuario',
'table' => 'password_resets',
'expire' => 60,
'throttle' => 60,
],
],
```

En el controlador del registro: app/Http/Controllers/Auth/RegisterController.php (observar que recibimos parámetro name y guardamos como nombre)

Lo que tiene establecido en la function create es:

```
return User::create([
'name' => $data['name'],
'email' => $data['email'],
'password' => Hash::make($data['password']),
]);
```

Si nos fijamos está dando la orden a la base de datos: User::create() así que se está creando un nuevo User en la tabla user con los campos: name, email,password (todos ellos dados por el usuario mediante el array $data). La orden create() recibe un array de datos php eso puede ser una

ventaja respecto al: save() de eloquent que tiene que recibir un objeto completo del modelo. También se observa que se guarda la password aplicándole un hash: Hash::make() En nuestro caso tenemos una tabla Usuario(id,nombre,password) id es autoincremental y no hace falta tocarlo, nombre nos lo envía el usuario con el parámetro: name, y password con el parámetro pertinente. Una vez creado el usuario resulta que el objeto del modelo obtenido: $usuario tiene ya registrada información en su campo id Así como en nuestra aplicación se entiende que un usuario puede desempeñar varios roles debemos guardar en la tabla: usuario_rol la información de roles. En este caso la línea: $rolUsuario = Rol::find(1); nos está diciendo que tomamos el primer rol (que es el de usuario) Esto es, para cada nuevo registro se le asignará el rol de usuario. Ya se encargará el administrador de agregarle otros roles.

Aprovechamos para ver como creamos directamente en la base de datos (sin usar el comando: save() del que ya hemos hablado) . Lo hacemos mediante el comando: create() al que se le pasa un array clave/valor que identifica el nombre del campo y el valor que va a tener el campo

```
protected function create(array $data)
{


        $usuario = ModelUsuario::create([
                //el formulario envía name y nosotros lo pasamos a ddbbnombre en DDBB
                'ddbbnombre' => $data['name'],
                //'email' => $data['email'],
                'ddbbclave' => Hash::make($data['password']),
        ]);

        $rolUsuario = Rol::find(1);
        /**
        * @var Usuario_rol nuevoUsuarioRol
        */
        $nuevoUsuarioRol = Usuario_rol::create(
        [ 'fk_usuario' => $usuario->idusuario, 'fk_rol' => $rolUsuario->idrol ]
        );
        return $usuario;
}
```

Observar que creamos un nuevo registro en la tabla usuario_rol con los ids correspondientes. Es casi como un insert sql

También hemos comentado el validador por si elegimos no estar enviado el campo email:

```
protected function validator(array $data)
{
return Validator::make($data, [
'name' => ['required', 'string', 'max:255'],
//'email' => ['required', 'string', 'email', 'max:255', 'unique:users'],
'password' => ['required', 'string', 'min:8', 'confirmed'],
]);
}
```

Ahora modificaremos **app/Http/Controllers/Auth/LoginController.php**:

```
class LoginController extends Controller
{
	public function username()
	{
		return 'ddbbnombre'; //el nombre del campo de la DDBB
	}
	public function logout()
	{
		session()->invalidate();
		session()->regenerate();
		return redirect('/');
	}
```

Vemos que hemos agregado un método username() que devuelve el nombre del campo en DDBB donde nosotros identificamos a nuestro usuario. Y un método logout() que elimina los datos de la sesión y vuelve a generar una nueva sesión: session()->invalidate();session()->regenerate();

Finalmente envía al usuario a la página raíz

Si usamos un nombre de campo en nuestra tabla de la base de datos diferente a: password debemos también crear: getAuthPassword() en la clase del modelo: ModelUsuario:

```
class ModelUsuario extends Authenticatable
{
protected $table = 'ddbbusuario';
public function getAuthPassword()
{
	return $this->ddbbclave;
}
```

vemos que devuelve el nombre del campo que corresponde con las passwords

Con los pasos que hemos descrito ya se puede crear usuario en la página de registro: http://localhost:8001/register (sustituir 8001 por el puerto en el que está arrancada la aplicación)

Pero luego en la página de login tenemos un problema aún: resources/views/auth/login.blade.php

Porque nosotros en el controlador pusimos:

```
$usuario = Usuario::create([
        //el formulario nos envía name y nosotros lo pasamos a nombre
        'nombre' => $data['name'],
        //'email' => $data['email'],
        'password' => Hash::make($data['password']),
]);
```

Así que el controlador ya no espera recibir un correo electrónico sino un nombre de usuario. Mientras que **login.blade.php** envía un email Debemos reemplazar el: <input id=”email” ….

por el input name que precisamos. El login.blade.php podría quedar así (es manifiestamente mejorable) En la imagen aparece marcado en azul la línea agregada y en verde la línea que se ha comentado

```
 9
10  <div class="card-body">
11      <form method="POST" action="{{ route('login') }}">
12          @csrf
13
14          <div class="form-group row">
15              <label for="email" class="col-md-4 col-form-label text-md-right">{{ __('Nombre usuario') }}</la
16
17              <div class="col-md-6">
18                  <input type="text" name="ddbbnombre" required autofocus>
19                  <!--
20                  <input id="email" type="email" class="form-control @error('email') is-invalid @enderror" na
21                  -->
22                  @error('email')
23                      <span class="invalid-feedback" role="alert">
24                          <strong>{{ $message }}</strong>
25                      </span>
26                  @enderror
27              </div>
28          </div>
29
```