**Curso laravel**

**Crear un nuevo proyecto**

* Primero ingresamos (primero creamos) en la carpeta donde vamos a crear nuestro proyecto con Laravel, desde la terminal.
* Ingresamos el comando (instalación con composer) **composer create-project laravel/laravel mi-proyecto-laravel** para crear un nuevo proyecto de laravel en la carpeta donde hemos ingresado anteriormente.
* También se puede instalar con el instalador de **laravel**: **composer global require "laravel/installer"** y luego  **laravel new proyecto-laravel**.
* La primera vista principal que veremos al iniciar un proyecto y abrirlo en el navegador, será el archivo **welcome.blade.php** alojado en la carpeta **resources**/**views**.
* Para visualizar el proyecto en local se emplea el comando **php artisan serve**.

**Rutas**

Laravel se basa en el patrón **front-controler**. Esto consiste en que vamos a tener un solo punto de entrada a nuestra aplicación, el archivo **public/index.php** al que se conoce como front-controler puesto que todas las peticiones de los usuarios a tu sitio web serán redirigidas a este archivo. Al tener un único punto de entrada los únicos archivos a los que el usuario va a poder acceder van a estar en los que se encuentren en la carpeta public. Por lo tanto, es donde irán alojados los **js** y **css**.

El archivo que se encarga de administrar las rutas es el archivo **routes/web.php**. Cuando un usuario escriba una url lo que va a hacer laravel es verificar si esa url la hemos definido en el archivo web.php

Para crear rutas se utiliza el siguiente código (en el archivo **routes/web.php**:

**Route::get('/', function () {**

**return view('welcome');**

**});**

Lo que quiere decir el código anterior es que en la raíz principal de la url, se abrirá la vista (alojada en **resources/views**) **welcome.blade.php**. Si en el get ponemos por ejemplo cursos, cuando ingresemos en la url **/cursos** se nos abrirá lo declarado en la función anónima.

Va a ver veces que vamos a necesitar mandar una variable por la url. Por ejemplo, si tenemos la url anterior /cursos, y tenemos dos cursos **/cursos/uno** y **/cursos/dos**, por lo que con solo un código de route, podemos hacer que nos redirija a una página diferente según a que curso ingresemos. El código route quedaría:

**Route::get('cursos/{curso}', function ($curso) {**

**return 'Bienvenido al curso: '. $curso;**

**});**

Por lo que si introducimos en la url: **cursos/curso2**, se mostrará el mensaje de: Bienvenido al curso: curso2.

Si queremos crear un route::get con una url que coincida con la función anterior pero que escriba un mensaje diferente al anterior (cursos/mensaje), hay que tener en cuenta que si queremos que no lo tome como una variable, habrá que escribir el código antes que el anterior, porque si no leerá primero la route con la variable que coincide con el segundo código y como laravel lee de arriba hacia abajo no la tendrá en cuenta a menos que lo escribamos antes. Por lo que esto estaría **correcto**:

**Route::get('cursos/mensaje', function () {**

**return 'Mensaje';**

**});**

**//Ruta con variables**

**Route::get('cursos/{curso}', function ($curso) {**

**return 'Bienvenido al curso: '. $curso;**

**});**

Y esto otro sería **incorrecto**:

**//Ruta con variables**

**Route::get('cursos/{curso}', function ($curso) {**

**return 'Bienvenido al curso: '. $curso;**

**});**

**Route::get('cursos/mensaje', function () {**

**return 'Mensaje';**

**});**

No tan solo puede recibir una variable, puede **recibir más de una variable**:

**Route::get('film/{categoria}/{titulo}', function ($categoria, $titulo) {**

**return 'La película <strong>' .$titulo. '</strong> es tipo ' .$categoria;**

**});**

Si queremos que una de las variables sea opcional, tan solo tendremos que añadir un símbolo de **?** al final de la variable en el get y decir en la función anónima que la variable opcional será de tipo **null** e indicar con un if si existe la categoría, se imprimirá un mensaje y si no existe (null) se imprimirá otro:

**Route::get('film/{titulo}/{categoria?}', function ($titulo, $categoria = null) {**

**if ($categoria) {**

**return 'La película <strong>' .$titulo. '</strong> es tipo ' .$categoria;**

**}else{**

**return 'Película: ' .$titulo;**

**}**

**});**

Esta es una manera de trabajar aunque no la mejor debido a que, en el archivo **web.php** solo debemos de manejar las rutas y todo lo que es la lógica (es decir los if) dejarlos para un **controlador**.

**Controladores**

Es el componente donde nos permite agrupar de una mejor manera las peticiones http, es el intermediario entre las vistas y los modelos, se encarga de realizar la lógica del negocio y controles necesarios de las solicitudes que llegan (ejemplo: peticiones a base de datos).

El archivo php que hace de controlador se encuentra en: **app/http/controllers/controller.php** o dentro de la carpeta **controllers** y crear nuevos controladores php.

Se pueden crear archivos desde la terminal, alojándonos en la carpeta de nuestro proyecto y colocando el comando **php artisan make:controller nombre\_controladorController** el controller al final es opcional, pero es para identificar que es un controlador. Y el comando php artisan make, se empleará en más de una ocasión para crear otro tipo de archivos.

Al insertar dicho comando (el que crea al controlador), se creará un archivo php que será utilizado como controlador. Al abrir dicho archivo creado aparecerá el siguiente código:

* **namespace App/Http/Controllers**: Define el lugar donde se encuentra el archivo (controlador creado).
* **use Illuminate/Http/Request**:
* **class nombreControlador extends Controller {}**: Aquí es donde escribiremos los métodos que se van a encargar de administrar nuestras rutas. La forma de asignar el control de las rutas (del archivo **routes/web.php**) al controlador creado es:
  + Especificar en el archivo **web.php** que vamos a utilizar dicho controlador de la siguiente manera: **use App\Http\Controllers\homeController;** Es decir indicamos la ruta donde se encuentra el controlador.
  + La ruta que teníamos especificada en el archivo **web.php**, por ejemplo, **Route::get('/', function () {**

**return view('welcome');**

**});**

Quedaría de la siguiente manera:

**Route::get('/', homeController::class);**

* + La clase del archivo del controlador creado quedaría de la siguiente manera (se utiliza el método mágico invoke):

**class homeController extends Controller**

**{**

**public function \_\_invoke(){**

**return view('welcome');**

**}**

**}**

Para rutas que tienen una misma ruta en común (por ejemplo cursos/php y cursos/Photoshop que tienen en común cursos) se crea un único controlador para esas rutas en vez de un controlador para cada tipo de ruta. Por lo que creamos otro controlador con los comandos anteriormente vistos (php artisan make:controller nombre\_controladorController). Cuando **el controlador controla más de una ruta ya no se emplea el método \_\_invoke**, se utilizan métodos con el nombre que nosotros queramos (por **convención** para las páginas principales se emplea **index**, para mostrar un curso **show** y para un formulario donde creamos datos de registro **create**). Por lo que quedaría:

**Route::get('cursos', [cursoController::class, 'index']);**

**Route::get('cursos/{curso}', [cursoController::class, 'create']);**

**Route::get('cursos/{titulo}/{categoria?}', [cursoController::class, 'show']);**

En el archivo web.php deberemos como antes con el otro controlador hemos hecho decir que vamos a usar un controlador con **use**, después de eso a las rutas que utilizan un controlador manejador de varias rutas, debemos de indicarle que además del controlador a utilizar, el método que se va a utilizar y esto se utiliza con un **array** como vemos en el código de arriba: **[cursoController::class, 'index']**.

El archivo del **controlador** quedaría de esta siguiente manera:

**class cursoController extends Controller**

**{**

**public function index(){**

**return 'Mensaje';**

**}**

**public function create($curso){**

**return 'Bienvenido al curso: '. $curso;**

**}**

**public function show($titulo, $categoria){**

**if ($categoria) {**

**return 'La película <strong>' .$titulo. '</strong> es tipo ' .$categoria;**

**}else{**

**return 'Película: ' .$titulo;**

**}**

**}**

**}**

Como se puede observar es casi lo mismo, que hemos visto sin el controlador solo que ahora es realizado como una arquitectura de tipo **MVC**, que es de la manera que **trabaja laravel**. Por lo que, dejamos la **lógica** al **controlador** y solo dejamos la url y el controlador a utilizar por esa ruta en el archivo de **rutas**. La estructura quedaría más limpia.

**Como quedaría el mismo código con otras versiones de laravel. Laravel 7:**

**Una ruta: Route::get('/', 'homeController');**

**Más de una ruta: Route::get('cursos/{curso}', 'cursoController@create');**

Y también habría que añadir los siguiente en el archivo **App/Providers/RouteServiceProvider.php** dentro de la clase **class RouteServiceProvider extends ServiceProvider**: **protected $namespace = ‘App\Http\Controllers’;**

Y además añadir lo siguiente en **public function boot en Route::middleware**:

**->namespace($this->namespace);**

Después de eso habría que quitar el use del archivo de las rutas.

**Grupos de ruta**

Hace referencia al conjunto de rutas que tienen una raíz en común. Como se vio en el apartado anterior, pueden existir rutas que tengan una misma raíz ruta en común. Ejemplo: cursos, cursos/mensaje, cursos/{variable}. Lo que realiza este grupo de rutas es que en vez de tener que ir añadiendo el mismo controlador a utilizar para cada ruta, se utiliza el método **Route::group(function(){});** Y a al método **group** le añadimos un método creado por nosotros al cuál le pasaremos el **controlador en común de todas las rutas**. De tal manera que el siguiente código:

Route::get('cursos', [cursoController::class, 'index']);

Route::get('cursos/{curso}', [cursoController::class, 'create']);

Quedaría:

**Route::controller(cursoController::class)->group(function(){**

**Route::get('cursos', 'index');**

**Route::get('cursos/{curso}', 'create');**

**});**

**Vistas**

Las vistas son los archivos que nos mostrará el contenido en forma de estructura web, es decir el html, estos archivos serán **blade.php** y se alojan en la carpeta **resources/views**. Por convención a las vistas s ele ponen el mismo nombre que los métodos utilizados en las rutas. Por ejemplo si en el archivo de rutas tenemos: Route::get('cursos/{curso}', 'create'); El archivo debería de llamarse: **create.blade.php**

Para retornar una vista, añadimos en el return del **controlador** el método **view**:

**public function \_\_invoke(){**

**return view('welcome');**

**}**

Si welcome estuviese dentro de otra carpeta se indicaría mediante la nomenclatura del punto de la siguiente manera: **carpeta.welcome**.

Si queremos pasarle una variable a la vista, deberemos de indicarlo en la función del controlador que hace referencia a esa vista mediante un array: **[‘curso’ =>$curso]**. Quedando como en el siguiente ejemplo:

**public function create($curso){**

**return view('cursos.create', [var => $curso]);**

**}**

Y en la **vista** donde queremos imprimir dicha variable, se podría ver de la siguiente manera (con **directiva Blade**):

**<h3 class=text-center>El curso es: {{$var}}></h3>**

Si queremos enviar **una variable que se llame igual a la variable que le pasamos a la función del controlador**, podemos utilizar el método **compact()** para ahorras código, ya que **compact(‘curso’)** es igual que poner **['curso' => $curso]**

**Si queremos que las vistas compartan algo en común** (como puede ser el header, el nav, el footer o una imagen de fondo) en vez de tener que aplicar los mismos cambios una y otra vez a cada vista, gracias a blade podemos solo declararlo una sola vez en un documento, para realizar esto deberemos de hacer lo siguiente:

* Crear una carpeta llamada **layouts** dentro de la carpeta **views** y un en el interior de la carpeta layouts crear un archivo de tipo blade.php. Éste será el archivo que sirva de plantilla para todas las demás vistas.
* Dentro del archivo creado en layouts, las partes que queremos que sean diferentes en otras páginas, utilizamos una **directiva blade** **@yield(‘nombreIdentificativo’)**.
* Las vistas que queremos que hereden esa plantilla deberemos escribir al inicio de la página: **@extends(carpeta.archivo)** Ejemplo: @extends(layouts.plantilla).
* En las páginas que queremos editar los yields de la plantilla insertamos las siguientes **directivas blade**:
  + **@section(‘title’, ‘Home’)**: Para una sola línea de código.
  + **@section('main content')**

**<h3 class=text-center>El curso es: <?php echo $curso?></h3>**

**@endsection**: Para más de una línea de código.

* Para que las **directivas Blades** funcionen, los archivos deben de ser tipo blade.php. Y el archivo quedaría así de limpio:

@extends('layouts.plantilla')

@section('title', 'Home')

@section('main content')

    <h3 class=text-center>El curso es: <?php echo $curso?></h3>

@endsection

**Generar dominio local (virtual host)**

* Primero ejecutamos el bloc de notas como administrador y luego le damos a **archivo** y en **abrir** y copiamos la siguiente ruta: **C:\Windows\System32\drivers\etc**
* Seleccionamos el archivo hosts (no aparecerá como documento de texto).
* Y pegamos el siguiente comando:  **127.0.0.1 prueba.test** y guardamos cambios.
* Ahora le tenemos que indicar a apache que cada vez que escribamos dicho dominio apunte al proyecto que deseamos.
* Abrimos la carpeta de apache/conf/extra y abrimos el archivo: **httpd-vhosts.conf**
* Copiamos dicho código al final del documento:

**NameVirtualHost \***

**<VirtualHost \*>**

**DocumentRoot "C:\xampp\htdocs"**

**ServerName localhost**

**</VirtualHost>**

* Y además este código, pero debemos de editar cual va a ser el proyecto que va a recibir dichos cambios y el nombre de dicho proyecto:

**<VirtualHost \*>**

**DocumentRoot "C:\miproyecto\httpdocs"**

**ServerName miproyecto.local.com**

**<Directory "C:\miproyecto\httpdocs">**

**Require all granted**

**</Directory>**

**</VirtualHost>**

**Introducción al manejo de base de datos MySQL**

Todo al respecto a cualquier configuración del proyecto se encuentra dentro de la carpeta **config**, en dicha carpeta encontramos el archivo para configurar las bases de datos **database.php**

Tendremos que darle las credenciales para conectarlo a la base de datos. Para dichas credenciales (como suelen tener datos sensibles como pueden ser contraseñas para conectarse a nuestra base de datos) se recomienda que se coloquen dentro de un archivo llamado **.env.** Esto es porque laravel ignora el archivo .env (ya que por defecto está añadido en el **.gitignore**)

Como podemos observar dentro del archivo de **database.php** existen llamadas al método **env($variable, ‘mysql’)**, esto significa lo siguiente:

* En primer lugar, va a buscar en el archivo .env una variable llamada **$variable**.
* Si no encuentra dicha variable, el valor por defecto será **mysql**.

**POR LO QUE EN RESUMEN EDITAREMOS EN EL ARCHIVO .ENV LAS CREDENCIALES DE MANERA GENERAL** exceptuando cuando tengamos que agregar una nueva credencial y variable, en cuyo caso serán añadidos a ambos archivos.

**Migraciones**

Carpeta **database/migrations**

En el primer archivo de dicha tabla (create users) encontramos una clase denominada **CreateUserTable** y se extiende de otra clase llamada **Migrations**.

Dentro de dicha clase encontramos dos métodos:

* **up**: dentro de dicho método se ejecuta un método de la clase S**chema** (**create**), dicho método create recibe como parámetro una **cadena** (la cuál será el nombre de la tabla que queremos crear), como segundo parámetro se le pasa una función anónima y dicha función anónima espera como parámetro un **objeto de tipo Blueprint**, dicho objeto se utiliza para crear las columnas de nuestra tabla. Dentro de la función anónima se llama varias veces a dicho objeto al cuál se le pasa distintos métodos propios del objeto. Dichos métodos disponen de unas características propias que se explican:

        Schema::create('users', function (Blueprint $table) {

            $table->id();//Integer Unsigned Increment

            $table->string('name');//varchar(255), se pueden especificar las medidas: $table->string('name', 100);

            //Si se quiere almacenar más tamaño de caracters sustituimo el string anterior por:

            //$table->text('name);

            $table->string('email')->unique();//Le pasa una propiedad ((unique) único).

            $table->timestamp('email\_verified\_at')->nullable();/\*Para guardar fechas (usado para verificación de correos electrónicos),

            con propiedad nullable que a la hora de realizar el registro este campo quedará vacío\*/

            $table->string('password');

            $table->rememberToken();//Columna tipo varchar de tamaño 100, se va almacenar cada vez que el usuario le de a mantener la sesión iniciada

            $table->timestamps();//Fecha y hora en la que se realizó dicho registro.

        });

* **down**: Llama a la clase **Schema** y ejecuta el método **dropIfExist** el cual elimina la tabla creada:

public function down()

    {

        Schema::dropIfExists('users');

    }

El segundo archivo de dicha tabla: **create password resset** sirve para resetear las contraseñas.

Para realizar las **migraciones** y crear las tablas de la base de datos, se utiliza el siguiente comando en la terminal del proyecto: **php artisan migrate** (ejecutará el método **up**).

**Crear migraciones**

Para crear una migración se emplea el comando por consola: **php artisan make:migration nombre\_migracion**. Esto nos creará un archivo dentro de la carpeta: **database/migrations**.

Nos debe de aparecer el nombre de la clase (si no es así lo cambiamos): **Class Cursos extends Migration.**

Para eliminar **la última tabla** que hemos creado, se emplea el comando: **php artisan migrate:rollback**

Cada vez que hagamos una nueva migración se sumará un lote más (batch), y cada vez que eliminemos se eliminará el lote último completo subido y se restará uno al batch.

Otra manera de crear una migración además de la ya vista es utilizando el comando en la teminal: **php artisan make:migration create\_nombremgracion\_table**. Si lo hacemos de esta manera nos podremos ahorrar tener que escribir los métodos de **Schema** de los métodos up y down.

Si **una vez hemos realizado las migraciones** queremos añadir más columnas o métodos o realizar cualquier tipo de cambio a los archivos de migraciones, para poder subir los cambios realizados, tendremos que insertar el siguiente comando: **php artisan migrate:fresh**. Esto recorre uno a uno las migraciones y ejecuta el método **down** (elimina las tablas) para después ejecutar el método **up** (y vuelve a subir las tablas) es una manera de **actualizar** tablas. **ES UNA MANERA, PERO NO ES LA MÁS RECOMENDABLE** ya que si desde la base de datos (por ejemplo phpmyadmin) ingresamos registros y luego en nuestro proyecto ejecutamos el **fresh**, esto tomará en cuenta lo que hay en nuestros archivos y no en la base de datos, por lo que eliminará los registros realizados.

**Modificar tablas con migraciones**

El comando: **php artisan migrate:refresh** realiza lo mismo que el método **fresh** solo que el método fresh eliminaba todas las tablas y el método **refresh** ejecuta uno a uno el método down() y luego el método up() uno a uno. Como se puede comprobar ambos métodos realizan lo mismo, pero hay una **diferencia entre el fresh y el refresh**: El método fresh no ejecuta el método down para eliminar las tablas y refresh si lo utiliza, por lo que si en el método down hay algún error de sintaxis u otra función especificada, no eliminará la tabla pero el fresh si lo hará ya que no tiene en cuenta el método down().

Para añadir una nueva columna a una tabla por medio de la terminal (**SIN SER DESTRUCTIVO POR SI EDITAMOS DATOS DIRECTAMENTE DESDE LA BASE DE DATOS**), se emplea el siguiente comando: **php artisan make:migration nombre\_tabla (por convención sería: add\_columnaNombre\_to\_nombreTabla\_table)**. Esta convención nos creará un archivo nuevo de migración pero, con una cierta estructura orientada a crear columnas para tablas, la estructura sería la siguiente:

**public function up()**

**{**

**Schema::table('users', function (Blueprint $table) {**

**//**

**});**

**}**

Nos crea el método up que llama a la clase Schema y ésta llama al método table. El método **table** es usado para modificar una tabla que ya existe. Este método recibe como parámetros, la tabla que vamos a modificar y una función anónima que recibe un objeto Blueprint. Dentro de la función anónima pondríamos lo siguiente:

$table->string('avatar')->nullable(); **Tenemos que pasarle un nullable, ya que si no nos dará error**.

**public function down()**

**{**

**Schema::table('users', function (Blueprint $table) {**

**$table->dropColumn('avatar');**

**});**

**}**

También nos creará el método down que recibirá también la clase Schema con el método table. Deberemos de pasarle la columna que se eliminará (con el método **dropColumn**) al ejecutarse el método down(). Una vez realizado todo esto indicamos el comando **php artisan migrate** para subir los cambios.

Esto hará que la columna añadida, se añada al final de la tabla. Si queremos **que la tabla a añadir se añada en una posición determinada de la tabla** realizamos el siguiente cambio en el método **up();** Le pasamos el método after() a la columna a añadir e indicamos dentro de dicho método después de que columna queremos que se añada la nueva, quedando el código: **$table->string('avatar')->nullable()->after('email');**

Si lo que queremos es modificar las propiedades de las columnas (por ejemplo la cantidad de varchar, cambiar que sea nullable, etc) deberemos de indicar primeramente el siguiente comando: **composer require doctrine/dbal**. Después creamos una migración para realizar los cambios de la propiedad de una columna: **php artisan make:migration cambiar\_propiedades\_to\_users\_table**. Dicha migración nos crea el mismo código que al crear la migración para crear una columna, lo único que deberemos de cambiar son los parámetros dentro del método up y método down, quedando:

**public function up()**

**{**

**Schema::table('users', function (Blueprint $table) {**

**$table->string('name', 150)->change();**

**});**

**}**

Le estamos indicando que cambie de 255 caracteres a solo 150 y le pasamos el método **change** para que nos realice los cambios. El método down:

public function down()

**{**

**Schema::table('users', function (Blueprint $table) {**

**$table->string('name', 255)->change();**

**});**

**}**

En el método down le indicamos que si queremos revertir los cambios (**rollback**) nos devuelva los caracteres que tenía antes de la modificación.

**ORM Eloquent CRUD**

Un **ORM** es un modelo de programación que permite mapear las estructuras de una base de datos relacional sobre una estructura lógica de entidades con el objetivo de simplificar y acelerar el desarrollo de nuestras aplicaciones. Las estructuras de la base de datos relacional quedan vinculadas con las entidades lógicas o base de datos virtual definida en el ORM, de tal modo que las acciones CRUD a ejecutar sobre la base de datos física se realizan de forma indirecta por medio del ORM. Es decir vamos a tratar a cada registro como si fuese un objeto. Esto **permite tener que evitar trabajar con consultas SQL**.

**Eloquent** hace uso de los Modelos para recibir o enviar la información a la base de datos. Por lo que, para trabajar con eloquent vamos a tener que crear un modelo: **php artisan make:model Model\_name(el mismo nombre que la tabla a la que vamos a relacionar pero en singular y la primera letra en mayúscula) (el nombre es una convención y esto permitirá dar a conocer a laravel sobre que tabla trabajar y otra opción es hacerlo todo manual dándole el nombre que nosotros queramos).** Para el nombre de la convención se pone en singular y automáticamente se relaciona con una tabla aunque ésta esté en plural, ejemplo: modelo User y tabla users. Pero esto solo funciona con palabras en inglés por lo que habría que tener cuidado con las tablas con nombres en español. (otra opción sería Nivele que lo relacionaría con la tabla niveles).

Los modelos serán alojados en: **app/models**

Al abrir el archivo que nos crea podemos observar que nos aparece lo siguiente:

**namespace App\Models;**

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

**use Illuminate\Database\Eloquent\Model;**

**class Curso extends Model**

**{**

**use HasFactory;**

**}**

El use Illuminate\Database\Eloquent\Model; nos sirve para acceder a todos los métodos de la clase Model (métodos CRUD) y también para tratar a los registros como objetos. A estos modelos normalmente los vamos a llamar desde los controladores.

Se usará bastante **tinker**. Tinker es una consola de comandos con la que podremos interactuar con todas las clases y métodos de nuestra aplicación, además de utilizar los comandos de **eloquent**. Para iniciar tinker: **php artisan tinker** (nos abre una power Shell dentro de la terminal) para salir de tinker: **exit**.

Para **crear un nuevo registro** debemos de crear un nuevo objeto, de la clase del modelo que estamos utilizando, en la power Shell de tinker: **use App\Models\Curso;** De esta manera apuntamos al modelo sobre el que queremos trabajar. Después se **crea una instancia de esa clase** (curso) = **$curso = new Curso;**

Una vez tengamos el objeto instanciado, **se le añaden propiedades y valores**, de la siguiente manera: **$curso->name = 'Laravel';** Si llamamos al objeto: **$curso;** nos devuelve el objeto con las propiedades y valores que hemos establecido.

Hasta aquí está el objeto creado, ahora lo que habría que realizar sería guardar los registros en la base de datos. Para que guarde los cambios habría que pasarle el método **save();** al objeto: **$curso->save();** Una vez realizado esto, si volvemos a llamar al obejto, nos devuelve además de los campos introducidos, los valores automáticos que se guardan en la tabla (como el id autoincremental, el create\_at o el update\_at).

Si queremos **cambiar el valor de una propiedad que ya hemos guardado** en la base de datos, se puede dar dos situaciones:

* Si existe el id, laravel reconocerá ese id y actualizará el registro con los valores actualizados.
* Si no existe el id, laravel creará un nuevo registro (objeto) ya que lo reconoce como otro tipo de registro.

**Para crear un modelo sin usar una convención**. Si queremos por ejemplo que el modelo Curso no administra a la tabla curso si no a otra tabla distinta. Para realizarlo debemos de indicar que emplee una tabla dentro de la clase en el modelo:

**class Curso extends Model**

**{**

**use HasFactory;**

**protected $table = 'users';**

**}**