**Regresar a una versión anterior del vagrant**

composer self-update --rollback

Enviar parámetros por rutas:

Route::get('(/miruta/{parametro}', [

'uses' => 'MiControladorController@Metodo'

]);

y en el controlador:

public function Metodo($parametro){}

# Eloquent ORM:

### Crear nuevos registros:

$user = User::create([

'name' => 'Luis Muñoz',

'email' => 'lmunoz@kobsa.com.pe',

'password' => bcrypt('123456'),

'sexo' => 'm'

]);

return 'Usuario guardado';

**¿Cómo configuramos qué campos se deben aceptar?**

En el modelo usando el arreglo $fillable:

protected $fillable = [

'name',

'email',

'password',

'sexo'

];

**Actualizar registros**

$user = User::find(1);

$user->gender = 'm';

$user->email = 'rickymuvel@gmail.com',

$user->save();

return "usuario actualizado";

Usando parámetros en Laravel

Route::get('/update/{id}', function($id) {

}

Obtener registros basados en un criterio

$users = User::where('gender', 'f')

->get();

**Seleccionar solo las columnas que queremos visualizar:**

$users = User::where('gender', 'f')

->get(['id', 'name', 'sexo']);

Seleccionar los ultimos diez usuarios registrados:

(Combinaremos condición y orden)

$users = User::orderBy('id', 'DESC')

->take(10)

->get();

return view('pages.home', compact('users'));

Eliminar registros de la base de datos:

$user = User::find($id);

$user->delete();

return view('pages.delete');

Método para llenar el control "Select" de los formularios:

$users = User::orderBy('name', 'ASC')

->lists('name', 'id');

Obtener el primer y ultimo elemento de nuestra tabla:

$first = User::first();

$all = User::all();

$last = $all->last();

**Paginación de registros**:

$users = User::orderBy('id', 'DESC')->paginate();

**Usando el borrado lógico (soft deletes):**

* Abrimos el modelo correspondiente (Modelo User en mi caso)
* Importamos la clase softDelete:

use Illuminate\Database\Eloquent\SoftDeletes;

* Usamos la clase dentro de la definición del modelo:

use SoftDeletes;

* Vamos a la migración correspondiente y añadimos el método *softDeletes();*

$table->softDeletes(); // esto añadirá el campo deleted\_at

**Obteniendo registros eliminados:**

**Obteniendo uno en particular:**

$user = User::withTrashed()->find(1);

**Obteniendo todos (Eliminados y no eliminados):**

User::withTrashed()->get();

**Restaurar un registro eliminado:**

$user = User::withTrashed()->find(1);

$user->restore();

**Eliminar un registro de forma permanente aún teniendo el softDeletes activo:**

Primero es necesario eliminarlo usando el método delete()

$user = User::find(1);

$user->delete()

$user = User::withTrashed()->find(1);

$user->forceDelete();

## Relaciones

Basado en el concepto de entidad relación, aquí podemos definir el tipo de relación que existe entre dos tablas. Esto lo hacemos porque en Laravel podemos obtener los elementos que se relacionan a una sola entidad con simplemente hacer la consulta. Por ejemplo, si tengo dos tablas, una es Oficina y otra Trabajador, yo podría decir que una oficina tiene muchos trabajadores (hasMany), pero un trabajador solo pertenece a una oficina (BelongsTo). la declaracion de las relaciones se hacen en la Clase Modelo correspondiente, de esta forma:

**Modelo:** ***Oficina***

class Oficina {

// nombre de la función en plural porque una oficina tiene muchos trabajadores

public function trabajadores(){

return $this->hasMany('Trabajador');

}

}

Ponemos el nombre del modelo con el cual está relacionada la clase oficina.

**Modelo: *Trabajador***

// nombre de la función en plural porque un trabajador pertenece a una oficina.

class Trabajador {

public function oficina(){

return $this->belongsTo('Oficina');

}

}

**¿Como obtener los trabajadores de una oficina con una simple busqueda?**

// En el controlador correspondiente podemos instanciar el modelo Oficina:

$oficinas = Oficina::get(); // Esto nos devolverá todo en una sola consulta.

// devolvemos esta info al cliente:

return response()->json(['oficinas', $oficinas], 201);

**Podemos extraer la información de la siguiente forma:**

data.datos.map(function(obj){

Oficina: {{obj.nombre}},

obj.trabajadores.map(function(ob){

Trabajador: {{ob.nombre}}

})

})

Es decir, en una sola consulta obtenemos la información relacionada sin necesidad de hacer JOINS ni procedimientos almacenados.

## Método Has()

**¿Qué ocurre cuando una oficina no tiene trabajadores asignados? (En caso que la oficina sea nueva por ejemplo)**

A la hora de hacer la consulta podemos usar el método has() de la siguiente forma:

$oficinas = Oficina::has('trabajadores')->get();

De esta forma solo se mostrarán los registros que tengan información relacionada. Ojo, este caso restringe los resultados a la relación existente entre los dos modelos. Si se desea mostrar si o sí será necesario usar el método anterior.

## Añadir Lógica al Modelo

**¿Cómo añadir lógica al modelo con el propósito de usarlo en la vista sin necesidad de hacer cálculos?**

Si por ejemplo necesitamos saber cuantos trabajadores tiene una oficina podemos crear en el mismo modelo Oficina un método que nos ayude a ese propósito:

// debajo de la función "trabajadores"

public function **get**NumTrabajadores**Attribute**(){

return count($this->trabajadores);

}

cuando lo usemos en la vista será necesario llamar solo a get**NumTrabajadores**Attribute con la parte **NumTrabajadores** en minúsculas y separadas por guión bajo: num\_trabajadores, así:

data.datos.**num\_trabajadores.**

### Método wereHas()

**¿Como condicionar o restringir aún más la salida de resultados en una consulta?**

Utilizaremos whereHas.

$oficinas = Oficinas::**whereHas**('trabajadores', function(**$query**){

**$query**->where('condicion', 'nuevo');

})->get();

De esta forma obtenemos las oficinas que tengan trabajadores nuevos. Sin embargo si tenemos 5 oficinas y en cada oficina hay un nuevo, nos mostrará las 5 oficinas. En caso quisieramos obtener las oficinas y los trabajadores nuevos, tendríamos que mejorar la consulta:

// Modificaremos el método getNumTrabajadoresAttribute

public function getNumTrabajadoresAttribute(){

return count($this->trabajadores->where('condicion', 'nuevo'));

}

// Nos totaliza los trabajadores nuevos.

// creamos un nuevo método:

public function getTrabajadoresNuevosAttribute(){

return $this->trabajadores->where('condicion', 'nuevo');

}

En la vista deberíamos examinar el siguiente arreglo:

data.datos.**trabajadores\_nuevos**

## Relación Muchos a Muchos

**¿Como hacemos la relación Muchos a Muchos?**

Si tuvieramos un modelo Usuario y otro Libro, podríamos decir que un autor tiene muchos libros, y un libro puede ser escrito por muchos autores. En ese sentido necesitamos una tabla intermedia donde podamos tener los ID de ambas tablas.

// Creamos la migración:

php artisan make:migration create\_libro\_usuario\_table

Ahí hacemos la relación de ambas tablas:

$table->increments('id');

$table->integer('libro\_id')->unsigned();

$table->integer('usuario\_id')->unsigned();

$table->foreign('libro\_id')->references('id')->on('libros')->**onDelete('cascade');**

$table->foreign('usuario\_id')->references('id')->on('usuarios')->**onDelete('cascade')**;

Ejecuta solo la migración de esa tabla:

php artisan migrate

Podemos crear un seeder para dicha tabla:

php artisan make:seeder LibroUsuarioTableSeeder

*Insertamos diez registros que relacionen al usuario y al libro con faker.*

AHORA SÍ, modificamos el modelo Usuario:

public function libros(){

return $this->belongsToMany('libro');

}

Lo mismo hacemos en el Modelo Libro:

public function usuarios(){

return $this->belongsToMany('usuario');

}

**Más sobre relaciones**

Tipos de relaciones:

hasOne -> relación uno a uno. Se lee como: Un usuario tiene un teléfono.

belongsTo -> le pertenece a. Se lee como: Un teléfono le pertenece a un usuario.

RELACIÓN DE UNO A UNO.

class User extends Model

{

/\*\*

\* Obtener el registro de teléfono asociado con el usuario

\*/

public function phone()

{

return $this->hasOne('App\Phone');

}

}

Una vez hecha la relación podemos obtener el registro utilizando las propiedades dinámicas de Eloquent:

$phone = User::find(1)->phone;

Eloquent determina la clave foránea de la relación basado en el nombre del modelo, en este se asume automáticamente que el modelo Phone tiene una clave foránea de nombre "user\_id". Si deseamos sobreescribir el nombre de la clave foránea es necesario especificarlo como segundo atributo:

return $this->hasOne('App\Phone', 'id\_usuario'); // por ejemplo, en caso la clave foránea sea id\_usuario.

Adicionalmente, Eloquent asume que la clave foránea debería tener un valor que coincida con la columna 'id' de la tabla padre. En otras palabras, si nuestra columna "id" es diferente a la asumida (id) entonces deberíamos especificarla como tercer parámetro:

return $this->hasOne('App\Phone', 'id\_usuario', 'mi\_id\_primario\_local');

DEFINIR LA INVERSA DE LA RELACIÓN

Podemos acceder al Phone desde la clase User:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model

{

/\*\*

\* Obtener el usuario dueño del teléfono

\*/

public function user()

{

return $this->belongsTo('App\User');

}

}

Del mismo modo podemos especificar los foreigns key y primary keys como lo hicimos en el primer caso:

/\*\*

\* Get the user that owns the phone.

\*/

public function user()

{

return $this->belongsTo('App\User', 'foreign\_key', 'other\_key');

}

RELACIÓN UNO A MUCHOS

En cierto modo es similar a la relación uno a uno, solo que en este caso usamos el método hasMany. Además si la clave foránea no se define siguiendo el formato sugerido por Laravel entonces tendremos dificultades para obtener la información. El formato de clave foranea es: "modelo\_id".

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

/\*\*

\* Obtener los comentarios de un blog del post.

\*/

public function comments()

{

return $this->hasMany('App\Comment');

}

}

Aquí también es importante mencionar que Eloquent determinará el formato de clave foránea por defecto (modelo\_id). Si el nombre de la clave foránea es diferente entonces será necesario definirla como segundo parámetro.

RELACIÓN UNO A MUCHOS (INVERSA)

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model

{

/\*\*

\* Obtener el post dueño del comentario

\*/

public function post()

{

return $this->belongsTo('App\Post', 'clave foranea si el formato no es el sugerido por Laravel');

}

}

La lectura es: Un comentario le pertenece a un Post y podemos según el comentario, obtener un post. Luego podemos ejecutar la consulta:

$comment = App\Comment::find(1);

echo $comment->post->title;

Nota:

En todos los casos será necesario definir la clave foranea por la cual ambas tablas o modelos están relacionados.