Laravel 5.2 : Bases de Datos. Eloquent ORM. Relaciones

Relaciones

Las tablas de las bases de datos están relacionadas entre sí: un post de un blog puede tener muchos comentarios, una orden puede estar relacionada con el usuario que la dio de alta. Eloquent facilita el trabajo con diferentes tipos de relaciones.

Definición de Relaciones

Las relaciones en Eloquent se definen como funciones en las clases modelo. Las relaciones sirven, pues, como potentes generadores de consultas. Definir relaciones como funciones proporciona métodos poderosos para encadenar consultas. Por ejemplo:

```
1 $user->posts()->where('active', 1)->get();
```

Tipos de relaciones:

One To One

Muy básica. Un modelo `User` puede estar relacionado con sólo un `Phone`. Para ello colocamos un método `phone` en el modelo `User`. El método `phone` ha de devolver los resultados del método `hasOne` en la clase modelo base de Eloquent:

```php
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Obtener el registro de
teléfono asociado con el usuario
     */
    public function phone()
    {
        return $this-
>hasOne('App\Phone');
    }
}
```

El primer argumento del método `hasOne` es el nombre del modelo relacionado. Definida la relación, podemos recuperar el registro relacionado con las propiedades dinámicas de Eloquent, que te permiten acceder a las funciones de relación como si fueran propiedades definidas en el modelo:

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

Eloquent crea el nombre de la clave foránea de la relación basándose en el nombre del modelo. En este caso el modelo `Phone` se

supone que tiene una clave foránea que se llama `user_id`. Si quieres cambiarlo, puedes pasarle un segundo argumento al método hasOne:

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

Además, Eloquent asume que la foreign key ha de tener un valor que coincide con la columna `id` (o la `$primaryKey`) del padre. En otras palabras, Eloquent va a buscar el valor de la columna `id` de la tabla user en la columna `user_id` del registro de phone. Si quieres que la relación use un valor distinto de `id`, se puede pasar un tercer argumento al método `hasOne` donde se especifique la clave primaria:

```
return $this->hasOne('App\Phone',
 'foreign_key', 'local_key');
```

Definición de la inversa de la relación

De momento podemos acceder al modelo `Phone` desde el modelo `User`. Ahora, definamos la relación inversa, en el modelo `Phone` que nos dejará acceder al `User` que es propietario del teléfono. Podemos definir la inversa de una relación hasOne con el método `belongsTo`:

```
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model
```

```php
class Phone extends Model
{
    /**
     * Devuelve el usuario
propietario de este telefono
     */
    public function user()
    {
        return $this-
>belongsTo('App\User');
    }
}
```

En el ejemplo de arriba, Eloquent hará coincidir el campo `user_id` del modelo Phone con un `id` del modelo User. Eloquent determina el nombre de la clave foránea por defecto examinando el nombre del método de la relación y añadiéndole el sufijo `_id`. No obstante, si la clave foránea en el modelo Phone no se llama `user_id`, se puede pasar como segundo argumento a `belongsTo` el nombre de la FK:

```php
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Devuelve el usuario
propietario de este telefono
     */
    public function user()
    {
        return $this-
>belongsTo('App\User',
'foreign_key');
    }
}
```

Y si la clave primaria del modelo padre no es

id, o quieres enlazar el modelo hijo a una columna diferente, se puede pasar un tercer argumento al método belognsTo que especifique la clave primaria de la tabla padre:

```php
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Devuelve el usuario
propietario de este telefono
     */
    public function user()
    {
        return $this-
>belongsTo('App\User',
'foreign_key', 'other_key');
    }
}
```

One To Many

Un post de un blog puede tener un infinito número de comentarios. Como todas las demás relaciones de Eloquent, la uno-a-muchos se define colocando un método en el modelo Eloquent:

```php
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Obtener los comentarios de
esta entrada del blog
     */
```

```
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Como siempre, Eloquent determinará automáticamente el nombre de la columna de la FK en la tabla Comment. Por convención, Eloquent tomará el *snake case* del modelo propietario y le añadirá `_id`. Así, para este ejemplo. Eloquent asumirá que la columna de la FK en la tabla comment es `post_id`.

Definida la relación, podemos acceder a la colacción de documentos mediante la propiedad comments.

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Por supuesto, como todas las relaciones sirven también como generadores de consultas, se pueden añadir restricciones a qué comentarios se recuperarán llamando al método `comments` y encadenando condiciones en la query:

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();
```

Como con el método `hasOne`, se pueden pasar nombres personalizados para la FK y la PK como argumentos adicionales:

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

```
        return $this-
        >hasMany('App\Comment',
        'foreign_key', 'local_key');
```

La relación inversa

Ahora podemos acceder a todos los comentarios desde el post. Definamos la relación que permita que un comentario acceda a su post padre. Para definir la inversa de hasMany, tenemos un método belongsTo:

```
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model

class Comment extends Model
{
    /**
     * Devuelve el post propietari
 de este comentario
     */
    public function post()
    {
        return $this-
>belongsTo('App\Post');
    }
}
```

Definida la relación, recuperamos el modelo Post para un Comment accediendo a su propiedad dinámica:

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

echo $comment->post->title;
```

Lo mismo que las otras relaciones sobre el nombre de la FK (post_id) y la PK en Post (id). Añadir métodos para definirlas si no se llaman así:

```php
/**
 * Devuelve el post propietario
de este comentario
 */
public function post()
{
    return $this-
>belongsTo('App\Post',
'foreign_key', 'other_key');
}
```

Many To Many

Son algo más complejas que las anteriores. Un usuario puede tener muchos roles y los roles son compartidos por muchos usuarios. Muchos usuarios tienen el rol "Admin". Para definir esta relación se necesitan tres tablas en la b.d.: `users`, `roles` y `role_user`. La tabla `role_user` es la tabla de relación, su nombre es la concatenación ordenada alfabéticamente de los nombres de las tablas que relaciona y contiene las columnas `user_id` y `role_id`.

La definiremos así en el modelo User:

```php
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Los roles a los que
pertenece el usuario
     */
    public function roles()
    {
        return $this-
>belongsToMany('App\Role');
    }
}
```

Con lo que podremos acceder a los roles de un usuario como un atributo más:

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    //
}
```

Igual que con las demás relaciones, podemos encadenar condiciones a la consulta:

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

Se aplica lo mismo que a las demás: eloquent intentará encontrar una tabla de relación que se llame como la concatenación con subrayados de los nombres de los modelos en orden alfabético, en este caso `roles_users`. Y además, las columnas que espera encontrar en la tabla de relación se llamarán `-modelo-_id` y `-otromodelo-_id`. Si no es así (caso normal), hay que pasar los nombres como parámetros:

```
return $this->belongsToMany('App\Role', 'user_rol', 'userId', 'rolId');
```

Relación inversa

También tenemos un método `belongsToMany` que se define en el otro modelo de la relación (el `Role`):

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model
```

```php

class Role extends Model
{
    /**
     * Los usuarios que pertenecen
a este rol
     */
    public function users()
    {
        return $this-
>belongsToMany('App\User');
    }
}
```

En este caso, la relación se define esactamente igual que en la parte User, y se le aplican las mismas normas de búsqueda de nombres de tablas de relación, columnas, etc.

Recuperar columnas de la tabla intermedia

Eloquent proporciona varios métodos para interactuar con la tabla intermedia de la relación. Sea que nuestro objeto User tiene varios objetos Role relacionados. Una vez definida la relación podemos acceder a la tabla intermedia usando el atributo pivot en los modelos:

```php
$user = App\User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot-
>created_at;
}
```

El atributo pivot contiene un modelo que representa a la tabla intermedia y puede usarse como cualquier otro modelo Eloquent.

Por defecto, sólo las columnas de relación

estarán presentes en el objeto pivot. Si la tabla de relacion contiene más columnas, hay que especificarlas al definir la relación:

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

Si queres que se mantengan automáticamente los timestamps `created_at` y `updated_at` en la tabla pivote, hay que especificar el método `withTimestamps` en la definición de la relación:

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

Filtrado de relaciones mediante columnas de la tabla intermedia

También podemos filtrar los resultados que devuelve `belongsToMany` usando los métodos `wherePivot` y `wherePivotIn` al definir la relación:

```
return $this->belongsToMany('App\Role')->wherePivot('approved', 1);

return $this->belongsToMany('App\Role')->wherePivotIn('approved', [1, 2]);
```

Has Many Through

Las relaciones has-many-through proporcionan un atajo adecuado para acceder a relaciones distantes vía una

relación intermedia. Por ejemplo, un modelo `Country` podría tener muchos modelos `Post` a través del modelo intermedio `User`. En este ejemplo, se pueden obtener todas las entradas de blog que pertenecen a un país. Veamos las tablas que se necesitan para definir esta relación:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

Aunque los posts no tienen una columna `country_id`, la relación `hasManyThrough` permite acceder a los posts de un country con `$country->posts`. Para ejecutar esta consulta, Eloquent inspeciona los `country_id` de la tabla intermedia (`users`). Tras encontrar los IDs que coinciden, se usan para consultar a la tabla posts.

La relación en el modelo `Country` sería:

```php
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model

class Country extends Model
{
    /**
     * Recupera todos los posts de
```

```
este país
     */
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

El primer argumento es al nombre del modelo final al que queremos acceder, y el segundo argumento, el modelo intermedio por el que pasamos para acceder al final.

Las convenciones de nombres son igual que en las demás relaciones. Para personalizar las claves de la relación se pueden pasar como argumentos tercero y cuarto; el tercero sería el nombre de la FK en el modelo intermedio y el cuarto el nombre de la FK en el modelo final:

```
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User', 'country_id', 'user_id');
    }
}
```

Relaciones polimórficas

Estructura de las tablas

Las relaciones polimórficas permiten que un modelo pertenezca a mas de un modelo en una sola asociación. Imaginemos que a los usuarios de tu aplicación les “gustan” tanto

los posts como los comentarios. Mediante relaciones polimórficas podemos usar una sola tabla `likes` para ambos escenarios. Veamos la estructura de las tablas:

```
posts
    id - integer
    title - string
    body - text

comments
    id - integer
    post_id - integer
    body - text

likes
    id - integer
    likeable_id - integer
    likeable_type - string
```

Las columnas clave son `likeable_id` y `likeable_type`. La `likeable_id` contiene el valor del ID del post o comentario, y la `likeable_type` contiene el nombre de clase del modelo en cuestion. La columna `likeable_type` es cómo el ORM determina qué tipo de modelo devolver cuando acceda a la relación:

Estructura de los modelos

```
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model

class Like extends Model
{
    /**
     * obtener todos los modelos
preferidos
     */
```

```
    public function likeable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * obtener todos los post
preferidos
     */
    public function likes()
    {
        return $this-
>morphMany('App\Like', 'likeable');
    }
}

class Comment extends Model
{
    /**
     * obtener todos los
comentarios preferidos
     */
    public function likes()
    {
        return $this-
>morphMany('App\Like', 'likeable');
    }
}
```

Recuperar relaciones polimórficas

Definidas las tablas y los modelos, podemos acceder a las relaciones desde los modelos: para acceder a todos los preferidos para un post, usamos la propiedad `likes`:

```
$post = App\Post::find(1);

foreach ($post->likes as $like) {
    //
}
```

También podemos recuperar el propietario de una relación polimórmica desde el modelo polimórfico mediante el método que

hace la llamada a morphTo. En este caso, este es el método likeable del modelo Like:

```
$like = App\Like::find(1);

$likeable = $like->likeable;
```

La relación likeable en el modelo Like devolverá un Post o un Comment, dependiendo del tipo de modelo que posee la preferencia.

Tipos polimórficos personalizados

Por defecto, Laravel usará el nombre completo cualificado para el tipo del modelo relacionado. En el ejemplo de arriba, en que un Like puede pertenecer a un Post o a un Comment, el linkeable_type por defecto puede ser App\Post ó App\Comment. Pero si queremos desacoplar la base de datos de la estructura interna de la aplicación, podríamos definir un "morph map" de la relación para instruir a Eloquent de que use el nombre de la tabla asociada con cada modelo en vez del nombre de la clase:

```
use
Illuminate\Database\Eloquent\Relati

Relation::morphMap([
    App\Post::class,
    App\Comment::class,
]);
```

O puedes especificar una cadena

personalizada para asociar con cada modelo:

```
use
Illuminate\Database\Eloquent\Relati
Relation::morphMap([
    'posts' => App\Post::class,
    'likes' => App\Like::class,
]);
```

Se puede registrar el `morphMap` en la función `boot` del `AppServiceProvider` o crear un proveedor de servicios separado para ello.

Relaciones polimórficas muchos-a-muchos

Estructura de tablas

Además de las relaciones polimórficas convencionales también podemos definirlas de muchos a muchos Por ejemplo, un `Post` y un modelo `Video` pueden compartir una relación polimórfica a un modelo `Tag`. Con una relación polimórfica muchos a muchos de este tipo podremos tener una única lista de tags únicos que están compartidos entre posts y videos. Primero, la estructura de tablas:

```
posts
    id - integer
    name - string

videos
```

```
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

Estructura de los modelos

Tanto `Post` como `Video` tienen un método `tag` que llama al método `morphToMany` de la clase base de Eloquent:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * obtener todos los tags del post
     */
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

La relación inversa

En el modelo `Tag`, hay que definir un método para cada uno de sus modelos relacionados. En este ejemplo, definiremos métodos

posts y videos:

```php
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    /**
     * dame todos los posts que
tienen asignada esta etiqueta
     */
    public function posts()
    {
        return $this-
>morphedByMany('App\Post',
'taggable');
    }

    /**
     * dame todos los videos que
tienen asignada esta etiqueta
     */
    public function videos()
    {
        return $this-
>morphedByMany('App\Video',
'taggable');
    }
}
```

Recuperar la relación

Podemos acceder a la relación desde los modelos. Para acceder a los tags de un post, se usa la propiedad tags:

```php
$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

También podemos recuperar el propietario de una relación polimórfica desde el modelo polimórfico accediendo al método que

ejecuta la llamada a morphedByMany. En nuestro caso, los métodos posts y videos del modelo Tag:

```
$tag = App\Tag::find(1);

foreach ($tag->videos as $video)
{
    //
}
```

Consultas a relaciones

Como todos los tipos de relaciones Eloquent se definen como funciones, podemos llamar a estas funciones para obtener una instancia de la relación sin tener que ejecutar las consultas de la relación. Además, todos los tipos de relaciones Eloquent también sirven como query builders, y nos permiten encadenar condiciones sobre la consulta de la relación antes de ejecutar realmente las consultas contra la b.d.

Sea un sistema de blog en que un modelo User tiene asociados muchos Posts:

```
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model

class User extends Model
{
    /**
     * dame todos los posts de este
 usuario
     */
    public function posts()
    {
        return $this-
>hasMany('App\Post');
    }
```

```
}
```

Podemos consultar esta relación `posts` y añadirle condiciones:

```php
$user = App\User::find(1);

$user->posts()->where('active', 1)->get();
```

Métodos de la relación vs. propiedades dinámicas

Si no necesitamos añadir restricciones a la consulta de relación, basta con acceder a la relación como una propiedad. Por ejemplo, en el mismo caso anterior, podemos acceder a todos los posts de un usuario vía:

```php
$user = App\User::find(1);

foreach ($user->posts as $post) {
    //
}
```

Las propiedades dinámicas se cargan en "*lazy loading*", sólo se cargarán los datos reales de la relación cuando se acceda realmente a ellos. Por esto, los desarrolladores suelen usar *eager loading* para precargar las relaciones que saben que se accederá a ellas tras cargar el modelo. Eager loading ejecuta muchas menos consultas SQL para cargar las mismas relaciones de un modelo.

Consultar la existencia de la propia relación

Cuando accedemos a los registros para un modelo, puedes querer limitar los resultados según la existencia o no de una relación. Por ejemplo, queremos recuperar

todos los posts que tienen al menos un comentario. Para ello, podemos pasar el nombre de la relación al método has:

```
// dame todos los posts que tienen al menos un comentario
$posts = App\Post::has('comments')->get();
```

Podemos especificar un operador y un número para ajustar más la consulta:

```
// dame todos los posts que tengan tres o más comentarios
$posts = Post::has('comments', '>=', 3)->get();
```

Los comandos has anidados pueden construirse con la notación de puntos. Por ejemplo: todos los posts que tengan al menos un comentario y un voto:

```
// Retrieve all posts that have at least one comment with votes...
$posts = Post::has('comments.votes')->get();
```

Aún más poderío, con los métodos whereHas y orWhereHas podemos poner condiciones where en las consultas de has, por ejemplo, por el contenido de un comentario:

```
// dame todos los posts que tengan al menos un comentario que contenga la palabra "foo%"
$posts = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

Carga Previa –

Eager Loading

Cuando accedemos a las relaciones Eloquent como **properties**, los datos de la relación se cargan en "**lazy loading**". Los datos de la relación no se cargarán hasta que se acceda a ellos por primera vez. Elquent también puede precargar las relaciones cuando se consulta al modelo padre. La carga previa evita el problema de las N+1 consultas. Para ilustrar el problema de las N+1 consultas imaginemos un modelo Book que está relacionado con Author:

```php
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * el autor que escribio el
libro
     */
    public function author()
    {
        return $this-
>belongsTo('App\Author');
    }
}
```

Recuperemos todos los libros y sus autores;

```php
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

Este bucle ejecutará una consulta para traerse todos los libros de la tabla y luego otra por cada libro para recuperar el autor. Así, si tenemos 25 libros, el bucle ejecutará 26 consultas.

Por suerte podemos usar la precarga para reducir esta operación a sólo dos consultas. Con el metodo with especificamos que la relación será eager loading:

```
$books =
App\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

Para esta operación sólo se ejecutan dos consultas:

```
select * from books

select * from authors where id in
 (1, 2, 3, 4, 5, ...)
```

> *Esto me parece un pasote. ¿Para qué hacer esa espantosa consulta con "in" que revienta todos los índices y carga tablas completas para nada? ¿No sería mejor utilizar una JOIN en la b.d. pare recuperar los registros de una sola vez? Como estoy estudiándome esto, esperaré a ver, pero desde luego si esto es así, me explico el criminal rendimiento de muchas de las aplicaciones que ves por ahí basadas en ORM, y me refuerza en mi convencimiento de que el viejo SQL que controlas tú mismo y en el que decides qué recuperar y cuándo sigue siendo la mejor técnica.*
>
> *Algo como: select b.*, a.name from books b, authors a where b.author_id = a.id*

Precarga de relaciones múltiples

A veces queremos precargar varias relaciones en una sola operación. Para ello siemplemente se pasan argumentos adicionales al método with:

```
$books = App\Book::with('author',
 'publisher')->get();
```

Carga previa anidada

Para precargar relaciones anidadas hay que usar la sintaxis de puntos. Por ejemplo, precarguemos todos los autores de los libros y todos los contactos personales de los autores sn un solo comando Eloquent:

```
$books =
App\Book::with('author.contacts')-
>get();
```

Restricción de las precargas

A veces queremos precargar una relación, restringiendo la consulta. Por ejemplo:

```
$users = App\User::with(['posts'
=> function ($query) {
    $query->where('title',
'like', '%first%');

}])->get();
```

En este ejemplo, Eloquent sólo precargará los posts tales que su columna title contenga la palabra first. Por supuesto, podemos llamar a otros métodos del query builder para personalizar más aún la operación de precarga.

```
$users = App\User::with(['posts'
=> function ($query) {
```

```
    $query->orderBy('created_at',
 'desc');

}])->get();
```

Post pre carga (Lazy Eager Loading

A veces queremos precargar una relación después de que el modelo padre ya se haya recuperado. Por ejemplo, esto puede ser útil si necesitas decidir dinámicamente si cargar o no los modelos relacionados:

```
$books = App\Book::all();

if ($someCondition) {
    $books->load('author',
'publisher');
}
```

Si queremos añadir condiciones a la consulta de precarga, podemos pasarle una Closure al método load:

```
$books->load(['author' =>
function ($query) {
    $query-
>orderBy('published_date',
'asc');
}]);
```

Inserción de modelos relacionados

El método save

Eloquent trae métodos convenientes para añadir nuevos modelos a las relaciones. Por

ejemplo, necesitamos insertar un nuevo Coment para un modelo Post. En vez de fijar manualmente el atributo `post_id` en el Comment, puedes insertar el Comment directamente desde el método save de la relación:

```
$comment = new
App\Comment(['message' => 'A new
comment.']);

$post = App\Post::find(1);

$post->comments()-
>save($comment);
```

No hemos accedido a la relación `comments` como una propiedad, sino que hemos llamado al método `comments` para obtener una instancia de la relación. El método save añadirá el `post_id` apropiado al nuevo modelo Comment.

Si necesitamos salvar múltiples modelos relacionados, puede usarse el método saveMany:

```
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' =>
 'A new comment.']),
    new App\Comment(['message' =>
 'Another comment.']),
]);
```

Save y las relaciones muchos a muchos

Cuando trabajamos con relaciones muchos a muchos el método save acepta un array de

atributos adicionales de la tabla intermedia como segundo argumento:

```
App\User::find(1)->roles()->save($role, ['expires' => $expires]);
```

El método Create

Además de los métodos save y saveMany, también tenemos el `create`, que acepta un array de atributos, crea un modelo y lo inserta en la b.d. De nuevo, la diferencia entre save y `create` es que save acepta una instancia modelo Eloquent completa y `create` acepta un array PHP plano:

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

Antes de usar el método `create`, hay que revisar bien la documentasión sobre la asignación masiva de atributos.

Actualización de relaciones “Belongs To”

Cuando actualizamos una relación `belongsTo`, podemos usar el método `associate`. Este método guarda la FK en el modelo hijo:

```
$account = App\Account::find(10);

$user->account()->
```

```
    >associate($account);

$user->save();
```

Cuando eliminamos una relación belongsTo, se usa el método dissociate. Este método eliminará la FK y la relación en el modelo hijo:

```
$user->account()->dissociate();

$user->save();
```

Relaciones muchos a muchos

Adjuntar, desadjuntar

Cuando trabajamos con relaciones muchos a muchos, Eloquent proporciona métodos de ayuda adicionales para facilitar el trabajo con los modelos relacionados. Por ejemplo, sea un usuario que puede tener muchos roles y un role que puede tener muchos usuarios. Para asignar un rol a un usuario insertando un registro en la tabla intermedia, se usa el método attach:

```
$user = App\User::find(1);

$user->roles()->attach($roleId);
```

Al adjuntar una relación a un modelo, se puede pasar también un array de datos adicionales que se insertarán en la tabla intermedia:

```
$user->roles()->attach($roleId,
['expires' => $expires]);
```

Para eliminar un rol de un usuario se usa el

método detach. Este método eliminará el registro adecuado de la tabla de relación, sin afectar, como es lógico a los modelos de ambos extremos.

```
// Eliminar un solo rol de un usuario
$user->roles()->detach($roleId);

// eliminar todos los roles del usuario.
$user->roles()->detach();
```

Attach y detach también aceptan arrays de IDs:

```
$user = App\User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([1 => ['expires' => $expires], 2, 3]);
```

Actualizar un registro de la tabla de relación

Se usa el método updateExistingPivot:

```
$user = App\User::find(1);

$user->roles()->updateExistingPivot($roleId, $attributes);
```

Sincronización de tabla de relación

Se puede usar el método sync para construir asociaciones muchos a muchos. El método sync acepta un array de IDs para colocar en la tabla intermedia. **Cualquier ID que no esté en el array se eliminará de la**

tabla intermedia. Por tanto, cuando se complete la operación, sólo quedarán en la tabla de relación los IDs del array que se ha pasado:

```
$user->roles()->sync([1, 2, 3]);
```

También pueden pasarse valores de la tabla intermedia con los IDs:

```
$user->roles()->sync([1 =>
['expires' => true], 2, 3]);
```

Actualizar los timestamps del padre

Cuando un modelo `belongsTo` o `belongsToMany` otros modelos, como un Comment que pertenece a un Post, es útil actualizar la fecha de actualización del padre cuando se actualiza el hijo. Por ejemplo, cuando se actualiza un modelo Comment, se puede actualizar el campo `updated_at` del Post propietario. Eloquent lo facilita. Añadir una propiedad `touches` que contenga los nombres de las relaciones al modelo hijo:

```
<?php

namespace App;

use
Illuminate\Database\Eloquent\Model

class Comment extends Model
{
    /**
     * Todas las relaciones que hay
 que actualizar
     *
     * @var array
     */
    protected $touches = ['post'];
```

```
17
18      /**
19       * Dame el post al que
20   pertenece este comentario
21       */
22      public function post()
23      {
            return $this-
        >belongsTo('App\Post');
        }
    }
```

Ahora, cuando actualizas un Comment, también se actualiza la columna `updated_at` en el Post propietario:

```
1  $comment = App\Comment::find(1);
2
3  $comment->text = 'Edit to this
4  comment!';
5
   $comment->save();
```