**MODULE – 9, 10(Forms, Controls)**

**1:Explain ORM [Object Relational Model ]**

[Introduction](https://laravel.com/docs/5.0/eloquent#introduction)

The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table.

Before getting started, be sure to configure a database connection in config/database.php.

## [Basic Usage](https://laravel.com/docs/5.0/eloquent#basic-usage)

To get started, create an Eloquent model. Models typically live in the app directory, but you are free to place them anywhere that can be auto-loaded according to your composer.json file. All Eloquent models extend Illuminate\Database\Eloquent\Model.

#### **Defining An Eloquent Model**

class User extends Model {}

**2:Explain - Eloquent Relationships:**

Database tables are often related to one another. For example, a blog post may have many comments, or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports several different types of relationships:

* [One To One](https://laravel.com/docs/5.1/eloquent-relationships#one-to-one)
* [One To Many](https://laravel.com/docs/5.1/eloquent-relationships#one-to-many)
* [Many To Many](https://laravel.com/docs/5.1/eloquent-relationships#many-to-many)
* [Has Many Through](https://laravel.com/docs/5.1/eloquent-relationships#has-many-through)
* [Polymorphic Relations](https://laravel.com/docs/5.1/eloquent-relationships#polymorphic-relations)
* [Many To Many Polymorphic Relations](https://laravel.com/docs/5.1/eloquent-relationships#many-to-many-polymorphic-relations)

## [Defining Relationships](https://laravel.com/docs/5.1/eloquent-relationships#defining-relationships)

Eloquent relationships are defined as functions on your Eloquent model classes. Since, like Eloquent models themselves, relationships also serve as powerful [query builders](https://laravel.com/docs/5.1/queries), defining relationships as functions provides powerful method chaining and querying capabilities. For example:

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

But, before diving too deep into using relationships, let's learn how to define each type:

### [**One To One**](https://laravel.com/docs/5.1/eloquent-relationships#one-to-one)

A one-to-one relationship is a very basic relation. For example, a User model might be associated with one Phone. To define this relationship, we place a phone method on the User model. The phone method should return the results of the hasOne method on the base Eloquent model class:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* Get the phone record associated with the user.

\*/

public function phone()

{

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

}

}

The first argument passed to the hasOne method is the name of the related model. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship functions as if they were properties defined on the model:

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

Eloquent assumes the foreign key of the relationship based on the model name. In this case, the Phone model is automatically assumed to have a user\_id foreign key. If you wish to override this convention, you may pass a second argument to the hasOne method:

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

Additionally, Eloquent assumes that the foreign key should have a value matching the id column of the parent. In other words, Eloquent will look for the value of the user's id column in the user\_id column of the Phone record. If you would like the relationship to use a value other than id, you may pass a third argument to the hasOne method specifying your custom key:

return $this->hasOne('App\Phone', 'foreign\_key', 'local\_key');

#### **Defining The Inverse Of The Relation**

So, we can access the Phone model from our User. Now, let's define a relationship on the Phone model that will let us access the User that owns the phone. We can define the inverse of a hasOne relationship using the belongsTo method:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model

{

/\*\*

\* Get the user that owns the phone.

\*/

public function user()

{

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

}

}

In the example above, Eloquent will try to match the user\_id from the Phone model to an id on the User model. Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with \_id. However, if the foreign key on the Phone model is not user\_id, you may pass a custom key name as the second argument to the belongsTo method:

/\*\*

\* Get the user that owns the phone.

\*/

public function user()

{

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

}

If your parent model does not use id as its primary key, or you wish to join the child model to a different column, you may pass a third argument to the belongsTo method specifying your parent table's custom key:

/\*\*

\* Get the user that owns the phone.

\*/

public function user()

{

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

}

### [**One To Many**](https://laravel.com/docs/5.1/eloquent-relationships#one-to-many)

A "one-to-many" relationship is used to define relationships where a single model owns any amount of other models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by placing a function on your Eloquent model:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

/\*\*

\* Get the comments for the blog post.

\*/

public function comments()

{

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

}

}

Remember, Eloquent will automatically determine the proper foreign key column on the Comment model. By convention, Eloquent will take the "snake case" name of the owning model and suffix it with \_id. So, for this example, Eloquent will assume the foreign key on the Comment model is post\_id.

Once the relationship has been defined, we can access the collection of comments by accessing the comments property. Remember, since Eloquent provides "dynamic properties", we can access relationship functions as if they were defined as properties on the model:

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

foreach ($comments as $comment) {

//

}

Of course, since all relationships also serve as query builders, you can add further constraints to which comments are retrieved by calling the comments method and continuing to chain conditions onto the query:

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

Like the hasOne method, you may also override the foreign and local keys by passing additional arguments to the hasMany method:

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

return $this->hasMany('App\Comment', 'foreign\_key', 'local\_key');

#### **Defining The Inverse Of The Relation**

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a hasMany relationship, define a relationship function on the child model which calls the belongsTo method:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model

{

/\*\*

\* Get the post that owns the comment.

\*/

public function post()

{

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

}

}

Once the relationship has been defined, we can retrieve the Post model for a Comment by accessing the post "dynamic property":

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

echo $comment->post->title;

In the example above, Eloquent will try to match the post\_id from the Comment model to an id on the Post model. Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with \_id. However, if the foreign key on the Comment model is not post\_id, you may pass a custom key name as the second argument to the belongsTo method:

/\*\*

\* Get the post that owns the comment.

\*/

public function post()

{

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

}

If your parent model does not use id as its primary key, or you wish to join the child model to a different column, you may pass a third argument to the belongsTo method specifying your parent table's custom key:

/\*\*

\* Get the post that owns the comment.

\*/

public function post()

{

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

}

### [**Many To Many**](https://laravel.com/docs/5.1/eloquent-relationships#many-to-many)

Many-to-many relations are slightly more complicated than hasOne and hasMany relationships. An example of such a relationship is a user with many roles, where the roles are also shared by other users. For example, many users may have the role of "Admin". To define this relationship, three database tables are needed: users, roles, and role\_user. The role\_user table is derived from the alphabetical order of the related model names, and contains the user\_id and role\_id columns.

Many-to-many relationships are defined by writing a method that calls the belongsToMany method on the base Eloquent class. For example, let's define the roles method on our User model:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* The roles that belong to the user.

\*/

public function roles()

{

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

}

}

Once the relationship is defined, you may access the user's roles using the roles dynamic property:

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

foreach ($user->roles as $role) {

//

}

Of course, like all other relationship types, you may call the roles method to continue chaining query constraints onto the relationship:

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

As mentioned previously, to determine the table name of the relationship's joining table, Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the belongsToMany method:

return $this->belongsToMany('App\Role', 'user\_roles');

In addition to customizing the name of the joining table, you may also customize the column names of the keys on the table by passing additional arguments to the belongsToMany method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to:

return $this->belongsToMany('App\Role', 'user\_roles', 'user\_id', 'role\_id');

#### **Defining The Inverse Of The Relationship**

To define the inverse of a many-to-many relationship, you simply place another call to belongsToMany on your related model. To continue our user roles example, let's define the users method on the Role model:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model

{

/\*\*

\* The users that belong to the role.

\*/

public function users()

{

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

}

}

As you can see, the relationship is defined exactly the same as its User counterpart, with the exception of simply referencing the App\User model. Since we're reusing the belongsToMany method, all of the usual table and key customization options are available when defining the inverse of many-to-many relationships.

#### **Retrieving Intermediate Table Columns**

As you have already learned, working with many-to-many relations requires the presence of an intermediate table. Eloquent provides some very helpful ways of interacting with this table. For example, let's assume our User object has many Role objects that it is related to. After accessing this relationship, we may access the intermediate table using the pivot attribute on the models:

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

foreach ($user->roles as $role) {

echo $role->pivot->created\_at;

}

Notice that each Role model we retrieve is automatically assigned a pivot attribute. This attribute contains a model representing the intermediate table, and may be used like any other Eloquent model.

By default, only the model keys will be present on the pivot object. If your pivot table contains extra attributes, you must specify them when defining the relationship:

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

If you want your pivot table to have automatically maintained created\_at and updated\_at timestamps, use the withTimestamps method on the relationship definition:

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

### [**Has Many Through**](https://laravel.com/docs/5.1/eloquent-relationships#has-many-through)

The "has-many-through" relationship provides a convenient short-cut for accessing distant relations via an intermediate relation. For example, a Country model might have many Post models through an intermediate User model. In this example, you could easily gather all blog posts for a given country. Let's look at the tables required to define this relationship:

countries

id - integer

name - string

users

id - integer

country\_id - integer

name - string

posts

id - integer

user\_id - integer

title - string

Though posts does not contain a country\_id column, the hasManyThrough relation provides access to a country's posts via $country->posts. To perform this query, Eloquent inspects the country\_id on the intermediate users table. After finding the matching user IDs, they are used to query the posts table.

Now that we have examined the table structure for the relationship, let's define it on the Country model:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model

{

/\*\*

\* Get all of the posts for the country.

\*/

public function posts()

{

return $this->hasManyThrough('App\Post', 'App\User');

}

}

The first argument passed to the hasManyThrough method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the hasManyThrough method. The third argument is the name of the foreign key on the intermediate model, while the fourth argument is the name of the foreign key on the final model.

class Country extends Model

{

public function posts()

{

return $this->hasManyThrough('App\Post', 'App\User', 'country\_id', 'user\_id');

}

}

### [**Polymorphic Relations**](https://laravel.com/docs/5.1/eloquent-relationships#polymorphic-relations)

#### **Table Structure**

Polymorphic relations allow a model to belong to more than one other model on a single association. For example, imagine you want to store photos for your staff members and for your products. Using polymorphic relationships, you can use a single photos table for both of these scenarios. First, let's examine the table structure required to build this relationship:

staff

id - integer

name - string

products

id - integer

price - integer

photos

id - integer

path - string

imageable\_id - integer

imageable\_type - string

Two important columns to note are the imageable\_id and imageable\_type columns on the photos table. The imageable\_id column will contain the ID value of the owning staff or product, while the imageable\_type column will contain the class name of the owning model. The imageable\_type column is how the ORM determines which "type" of owning model to return when accessing the imageable relation.

#### **Model Structure**

Next, let's examine the model definitions needed to build this relationship:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Photo extends Model

{

/\*\*

\* Get all of the owning imageable models.

\*/

public function imageable()

{

return $this->morphTo();

}

}

class Staff extends Model

{

/\*\*

\* Get all of the staff member's photos.

\*/

public function photos()

{

return $this->morphMany('App\Photo', 'imageable');

}

}

class Product extends Model

{

/\*\*

\* Get all of the product's photos.

\*/

public function photos()

{

return $this->morphMany('App\Photo', 'imageable');

}

}

#### **Retrieving Polymorphic Relations**

Once your database table and models are defined, you may access the relationships via your models. For example, to access all of the photos for a staff member, we can simply use the photos dynamic property:

$staff = App\Staff::find(1);

foreach ($staff->photos as $photo) {

//

}

You may also retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to morphTo. In our case, that is the imageable method on the Photo model. So, we will access that method as a dynamic property:

$photo = App\Photo::find(1);

$imageable = $photo->imageable;

The imageable relation on the Photo model will return either a Staff or Product instance, depending on which type of model owns the photo.

#### **Custom Polymorphic Types**

By default, Laravel will use the fully qualified class name to store the type of the related model. For instance, given the example above where a Like may belong to a Post or a Comment, the default likable\_type would be either App\Post or App\Comment, respectively. However, you may wish to decouple your database from your application's internal structure. In that case, you may define a relationship "morph map" to instruct Eloquent to use the table name associated with each model instead of the class name:

Relation::morphMap([

App\Post::class,

App\Comment::class,

]);

Or, you may specify a custom string to associate with each model:

Relation::morphMap([

'posts' => App\Post::class,

'likes' => App\Like::class,

]);

You may register the morphMap in your AppServiceProvider or create a separate service provider if you wish.

### [**Many To Many Polymorphic Relations**](https://laravel.com/docs/5.1/eloquent-relationships#many-to-many-polymorphic-relations)

#### **Table Structure**

In addition to traditional polymorphic relations, you may also define "many-to-many" polymorphic relations. For example, a blog Post and Video model could share a polymorphic relation to a Tag model. Using a many-to-many polymorphic relation allows you to have a single list of unique tags that are shared across blog posts and videos. First, let's examine the table structure:

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

#### **Model Structure**

Next, we're ready to define the relationships on the model. The Post and Video models will both have a tags method that calls the morphToMany method on the base Eloquent class:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

/\*\*

\* Get all of the tags for the post.

\*/

public function tags()

{

return $this->morphToMany('App\Tag', 'taggable');

}

}

#### **Defining The Inverse Of The Relationship**

Next, on the Tag model, you should define a method for each of its related models. So, for this example, we will define a posts method and a videos method:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model

{

/\*\*

\* Get all of the posts that are assigned this tag.

\*/

public function posts()

{

return $this->morphedByMany('App\Post', 'taggable');

}

/\*\*

\* Get all of the videos that are assigned this tag.

\*/

public function videos()

{

return $this->morphedByMany('App\Video', 'taggable');

}

}

#### **Retrieving The Relationship**

Once your database table and models are defined, you may access the relationships via your models. For example, to access all of the tags for a post, you can simply use the tags dynamic property:

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

foreach ($post->tags as $tag) {

//

}

You may also retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to morphedByMany. In our case, that is the posts or videos methods on the Tag model. So, you will access those methods as dynamic properties:

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

foreach ($tag->videos as $video) {

//

}

## [Querying Relations](https://laravel.com/docs/5.1/eloquent-relationships#querying-relations)

Since all types of Eloquent relationships are defined via functions, you may call those functions to obtain an instance of the relationship without actually executing the relationship queries. In addition, all types of Eloquent relationships also serve as [query builders](https://laravel.com/docs/5.1/queries), allowing you to continue to chain constraints onto the relationship query before finally executing the SQL against your database.

For example, imagine a blog system in which a User model has many associated Post models:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* Get all of the posts for the user.

\*/

public function posts()

{

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

}

}

You may query the posts relationship and add additional constraints to the relationship like so:

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

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

Note that you are able to use any of the [query builder](https://laravel.com/docs/5.1/queries) methods on the relationship!

#### **Relationship Methods Vs. Dynamic Properties**

If you do not need to add additional constraints to an Eloquent relationship query, you may simply access the relationship as if it were a property. For example, continuing to use our User and Post example models, we may access all of a user's posts like so:

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

foreach ($user->posts as $post) {

//

}

Dynamic properties are "lazy loading", meaning they will only load their relationship data when you actually access them. Because of this, developers often use [eager loading](https://laravel.com/docs/5.1/eloquent-relationships#eager-loading) to pre-load relationships they know will be accessed after loading the model. Eager loading provides a significant reduction in SQL queries that must be executed to load a model's relations.

#### **Querying Relationship Existence**

When accessing the records for a model, you may wish to limit your results based on the existence of a relationship. For example, imagine you want to retrieve all blog posts that have at least one comment. To do so, you may pass the name of the relationship to the has method:

// Retrieve all posts that have at least one comment...

$posts = App\Post::has('comments')->get();

You may also specify an operator and count to further customize the query:

// Retrieve all posts that have three or more comments...

$posts = Post::has('comments', '>=', 3)->get();

Nested has statements may also be constructed using "dot" notation. For example, you may retrieve all posts that have at least one comment and vote:

// Retrieve all posts that have at least one comment with votes...

$posts = Post::has('comments.votes')->get();

If you need even more power, you may use the whereHas and orWhereHas methods to put "where" conditions on your has queries. These methods allow you to add customized constraints to a relationship constraint, such as checking the content of a comment:

// Retrieve all posts with at least one comment containing words like foo%

$posts = Post::whereHas('comments', function ($query) {

$query->where('content', 'like', 'foo%');

})->get();

### [**Eager Loading**](https://laravel.com/docs/5.1/eloquent-relationships#eager-loading)

When accessing Eloquent relationships as properties, the relationship data is "lazy loaded". This means the relationship data is not actually loaded until you first access the property. However, Eloquent can "eager load" relationships at the time you query the parent model. Eager loading alleviates the N + 1 query problem. To illustrate the N + 1 query problem, consider a Book model that is related to Author:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model

{

/\*\*

\* Get the author that wrote the book.

\*/

public function author()

{

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

}

}

Now, let's retrieve all books and their authors:

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

foreach ($books as $book) {

echo $book->author->name;

}

This loop will execute 1 query to retrieve all of the books on the table, then another query for each book to retrieve the author. So, if we have 25 books, this loop would run 26 queries: 1 for the original book, and 25 additional queries to retrieve the author of each book.

Thankfully, we can use eager loading to reduce this operation to just 2 queries. When querying, you may specify which relationships should be eager loaded using the with method:

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

foreach ($books as $book) {

echo $book->author->name;

}

For this operation, only two queries will be executed:

select \* from books

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

#### **Eager Loading Multiple Relationships**

Sometimes you may need to eager load several different relationships in a single operation. To do so, just pass additional arguments to the with method:

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

#### **Nested Eager Loading**

To eager load nested relationships, you may use "dot" syntax. For example, let's eager load all of the book's authors and all of the author's personal contacts in one Eloquent statement:

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

### [**Constraining Eager Loads**](https://laravel.com/docs/5.1/eloquent-relationships#constraining-eager-loads)

Sometimes you may wish to eager load a relationship, but also specify additional query constraints for the eager loading query. Here's an example:

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

$query->where('title', 'like', '%first%');

}])->get();

In this example, Eloquent will only eager load posts that if the post's title column contains the word first. Of course, you may call other [query builder](https://laravel.com/docs/5.1/queries) to further customize the eager loading operation:

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

$query->orderBy('created\_at', 'desc');

}])->get();

### [**Lazy Eager Loading**](https://laravel.com/docs/5.1/eloquent-relationships#lazy-eager-loading)

Sometimes you may need to eager load a relationship after the parent model has already been retrieved. For example, this may be useful if you need to dynamically decide whether to load related models:

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

if ($someCondition) {

$books->load('author', 'publisher');

}

If you need to set additional query constraints on the eager loading query, you may pass a Closure to the load method:

$books->load(['author' => function ($query) {

$query->orderBy('published\_date', 'asc');

}]);

## [Inserting Related Models](https://laravel.com/docs/5.1/eloquent-relationships#inserting-related-models)

#### **The Save Method**

Eloquent provides convenient methods for adding new models to relationships. For example, perhaps you need to insert a new Comment for a Post model. Instead of manually setting the post\_id attribute on the Comment, you may insert the Comment directly from the relationship's save method:

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

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

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

Notice that we did not access the comments relationship as a dynamic property. Instead, we called the comments method to obtain an instance of the relationship. The save method will automatically add the appropriate post\_id value to the new Comment model.

If you need to save multiple related models, you may use the saveMany method:

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

$post->comments()->saveMany([

new App\Comment(['message' => 'A new comment.']),

new App\Comment(['message' => 'Another comment.']),

]);

#### **Save & Many To Many Relationships**

When working with a many-to-many relationship, the save method accepts an array of additional intermediate table attributes as its second argument:

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

#### **The Create Method**

In addition to the save and saveMany methods, you may also use the create method, which accepts an array of attributes, creates a model, and inserts it into the database. Again, the difference between save and create is that save accepts a full Eloquent model instance while create accepts a plain PHP array:

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

$comment = $post->comments()->create([

'message' => 'A new comment.',

]);

Before using the create method, be sure to review the documentation on attribute [mass assignment](https://laravel.com/docs/5.1/eloquent#mass-assignment).

#### [**Updating "Belongs To" Relationships**](https://laravel.com/docs/5.1/eloquent-relationships#updating-belongs-to-relationships)

When updating a belongsTo relationship, you may use the associate method. This method will set the foreign key on the child model:

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

$user->account()->associate($account);

$user->save();

When removing a belongsTo relationship, you may use the dissociate method. This method will reset the foreign key as well as the relation on the child model:

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

$user->save();

### [**Many To Many Relationships**](https://laravel.com/docs/5.1/eloquent-relationships#inserting-many-to-many-relationships)

#### **Attaching / Detaching**

When working with many-to-many relationships, Eloquent provides a few additional helper methods to make working with related models more convenient. For example, let's imagine a user can have many roles and a role can have many users. To attach a role to a user by inserting a record in the intermediate table that joins the models, use the attach method:

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

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

When attaching a relationship to a model, you may also pass an array of additional data to be inserted into the intermediate table:

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

Of course, sometimes it may be necessary to remove a role from a user. To remove a many-to-many relationship record, use the detach method. The detach method will remove the appropriate record out of the intermediate table; however, both models will remain in the database:

// Detach a single role from the user...

$user->roles()->detach($roleId);

// Detach all roles from the user...

$user->roles()->detach();

For convenience, attach and detach also accept arrays of IDs as input:

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

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

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

#### **Syncing For Convenience**

You may also use the sync method to construct many-to-many associations. The sync method accepts an array of IDs to place on the intermediate table. Any IDs that are not in the given array will be removed from the intermediate table. So, after this operation is complete, only the IDs in the array will exist in the intermediate table:

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

You may also pass additional intermediate table values with the IDs:

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

### [**Touching Parent Timestamps**](https://laravel.com/docs/5.1/eloquent-relationships#touching-parent-timestamps)

When a model belongsTo or belongsToMany another model, such as a Comment which belongs to a Post, it is sometimes helpful to update the parent's timestamp when the child model is updated. For example, when a Comment model is updated, you may want to automatically "touch" the updated\_at timestamp of the owning Post. Eloquent makes it easy. Just add a touches property containing the names of the relationships to the child model:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model

{

/\*\*

\* All of the relationships to be touched.

\*

\* @var array

\*/

protected $touches = ['post'];

/\*\*

\* Get the post that the comment belongs to.

\*/

public function post()

{

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

}

}

Now, when you update a Comment, the owning Post will have its updated\_at column updated as well:

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

$comment->text = 'Edit to this comment!';

$comment->save();

**4: What is Eager Loading and Lazy Loading ?**

## What is Eager Loading?

When you are fetching any models from the database and then doing any type of processing on the model’s relations, it’s important that you use [eager loading](https://laravel.com/docs/8.x/eloquent-relationships#eager-loading). Eager loading is super simple using Laravel and basically prevents you from encountering the N+1 problem with your data. This problem is caused by making N+1 queries to the database, where N is the number of items being fetched from the database. To explain this better and give it some context, let's check out the example below.

Imagine that you have two models (Comment and Author) with a one-to-one relationship between them. Now imagine that you have 100 comments and you want to loop through each one of them and output the author’s name.

Without eager loading, your code might look like this:

1$comments = Comment::all();

2

3foreach ($comments as $comment ) {

4 print\_r($comment->author->name);

5}

The code above would result in 101 database queries because it the results are "lazy loaded"! The first query would be to fetch all of the comments. The other one hundred queries would come from getting the author’s name in each iteration of the loop. Obviously, this can cause performance issues and slow down your application. So, how would we improve this?

By using eager loading, we could change the code to say:

1$comments = Comment::with(‘authors’)->get();

2

3foreach ($comments as $comment ) {

4 print\_r($comment->author->name);

5}

As you can see, this code looks almost the same and is still readable. By adding the ::with('authors') this will fetch all of the comments and then make another query to fetch the authors at once. So, this means that we will have cut down the query from **101** to **2**!

# Lazy Loading :

Generally we can say, Lazy Loading works on a website or webpage for delaying load or initialization of resources or objects.  
Example when we visit facebook its load first time show us some gray images then its load the our timeline.

## Lazy Loading : In Laravel Lazy Loading can be seen on two ways but both are nuance.

Let us suppose a relationship: Users and Posts.

**a User can have multiple Posts**  
**a Post can have single User**  
Here we can say User is parent table for Posts.  
Posts is the child table for Users table.

$users = User::get();

Here we are getting all the users. Now when we loop through the users and accessing the post of that user we are doing lazy loading.

Below syntax is just for visualization $user(3) its not allowed in php.

foreach($users(3) as $user){

$user->post;

}

Dynamic relationship properties perform "lazy loading", meaning they will only load their relationship data when you actually access them.  
Here we find "N + 1" query problem.  
suppose we have three users so **N** = $user =User->get() = 3 collections

foreach($users as $user){

$user->post;

}

The above loop will loop for three times. N = 3. $user->post query will be called three times so we have three queries and a single query for getting the $users = User->get().  
User->get() = 1  
$user->post = N times. so we have N+1 problem.

## Lazy Loading: This can be explain when we do relationship from child to parent table.

$posts = Post::get().  
To get the **user** for every **post** we will do loop.  
Post is child table for User parent table. Here we also find "N + 1" query problem.

To alleviate N+1 we use eager-loading in laravel. The with() method give us the power to do eager-loading.

$users = User::with("posts")->get();

behind the scene the with("posts") do a where...in.. query on mysql.

# Lazy Collection in laravel or overall concept:

Lazy collection works for a big dataset like multi gigabytes file sizes. For Example, if you want to read data from a big size file it can exceed your memory size. SO lazy collections can be used for your reading the data from your file in small parts.

Laravel **cursor** method on Eloquent ORM give us the lazy collection. The cursor behind the scene uses the PHP generator.

#### Laravel cursor gives us lazy collections

#### LazyCollection::make(function () { } ) also create lazy collecitons

#### Flight::lazy() also give us lazy collections.

the lazy method returns a flattened LazyCollection of Eloquent models.

What is difference between lazy loading and eager loading?

Lazy Loading vs. Eager Loading. While **lazy loading delays the initialization of a resource, eager loading initializes or loads a resource as soon as the code is executed**.