Relationship

One To One ::

In User model :: (Parent)	In phone Model:: (Child)

class User extends Model
{
public function phone()
{
return $this->hasOne('App\Phone');
}
}
	class Phone extends Model
	{
	public function user()
	{
	return $this->belongsTo('App\User');
	}
	}

model is automatically assumed to have a `user_id` foreign key. To override this convention, pass a second argument to the method:

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

Eloquent will look for the value of the user's `id` column in the `user_id` column of the `Phone` record. To use a value other than `id`, pass a third argument to the `hasOne` method specifying custom key:

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

Retrieve the info:: `$phone = User::find(1)->phone;`

One To Many ::

class Post extends Model
{
public function comments()
{
return $this->hasMany('App\Comment');
}
}
	class Comment extends Model
	{
	public function post()
	{
	return $this->belongsTo('App\Post');
	}
	}

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->belongsTo('App\Post', 'foreign_key');

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

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

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

Or,

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

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

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

Many To Many ::

```
class User extends Model
{
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

```
class Role extends Model
{
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

Retrieve the info : :

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

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

Or,

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

Eloquent will join the two related model names in alphabetical order, to **override** this convention

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

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

```
return $this->belongsToMany('App\Role', 'role_user', 'user_id', 'role_id');
```

many-to-many relations requires the presence of an intermediate table, 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;
}
```

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();
```

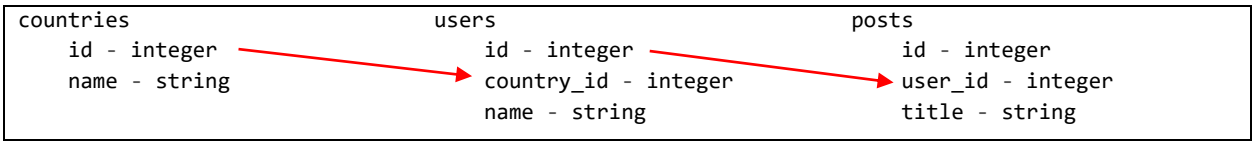
filter the results returned by `belongsToMany` using the `wherePivot` and `wherePivotIn`methods when defining the relationship:

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

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

Has Many Through

Convenient short-cut for accessing distant relations via an intermediate relation.



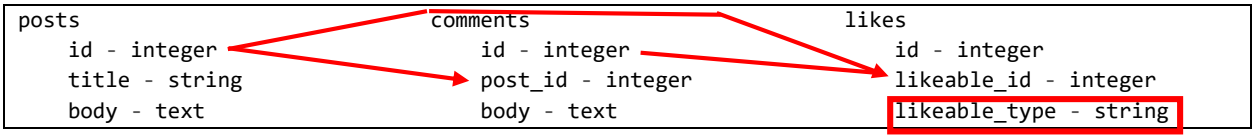

```
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

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.

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, the fourth argument is the name of the foreign key on the final model, and the fifth argument is the local key:

```
return $this->hasManyThrough('App\Post', 'App\User', 'country_id', 'user_id', 'id');
```

Polymorphic Relations ::




```
class Post extends Model
{
    public function likes()
    {
        return $this->morphMany('App\Like', 'likeable');
    }
}
```

```
class Comment extends Model
{
    public function likes()
    {
        return $this->morphMany('App\Like', 'likeable');
    }
}
```

```
class Like extends Model
{
    public function likeable()
    {
        return $this->morphTo();
    }
}
```

Access the Value::

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

foreach ($post->likes as $like) {                    //access all of the likes for a post

    //
```

```
}
```

retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to `morphTo`

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

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

By default, Laravel will use the fully qualified class name to store the type of the related model. 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:

```
use Illuminate\Database\Eloquent\Relations\Relation;
Relation::morphMap([
    App\Post::class,
    App\Comment::class,
]);
```

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

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

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

Many To Many Polymorphic Relations

```
posts                  videos                   tags                    taggables
    id - integer           id - integer             id - integer ------> tag_id - integer
    name - string          name - string            name - string        taggable_id - integer
                                                                         taggable_type - string
```

```
class Post extends Model

{

    public function tags()

    {

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

    }

}
```

```
class Video extends Model

{

    public function tags()

    {

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

    }

}
```

```
class Tag extends Model                              //Defining The Inverse Of The Relationship

{
```

```
    public function posts()
    {    return $this->morphedByMany('App\Post', 'taggable');    }

    public function videos()
    {    return $this->morphedByMany('App\Video', 'taggable');    }
}
```

Retrieving the Value::

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

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

Retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to `morphedByMany`

```
$tag = App\Tag::find(1);
foreach ($tag->videos as $video) {
    //
}
```

Querying Relations

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

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

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.

```
$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.

Querying Relationship Existence::

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

$posts = Post::has('comments', '>=', 3)->get();// Retrieve all posts that have three or more comments

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

// 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 ::

Relationship data is "lazy loaded". This means the relationship data is not actually loaded until you first access the property. Eager loading alleviates the N + 1 query problem.

```
class Book extends Model
{
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
==========================================
$books = App\Book::all();
foreach ($books as $book) {
    echo $book->author->name;
}
```

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

foreach ($books as $book) {

    echo $book->author->name;

}
```

```
$books = App\Book::with('author', 'publisher')->get();              //with Multiple Relationship
$books = App\Book::with('author.contacts')->get();                  // Nested Eager Loading

$users = App\User::with(['posts' => function ($query) {             //Constraining Eager Loads
                    $query->where('title', 'like', '%first%');
                    $query->orderBy('created_at', 'desc');

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

Lazy Eager Loading

eager load a relationship after the parent model has already been retrieved. if you need to dynamically decide whether to load related models:

```
$books = App\Book::all();
if ($someCondition) {
    $books->load('author', 'publisher');
}
```

set additional query constraints on the eager loading query

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

Inserting Related Models::

```
$comment = new App\Comment(['message' => 'A new comment.']);
$post = App\Post::find(1);
$post->comments()->save($comment);


$post = App\Post::find(1);                  //save multiple related models
$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

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

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.',
]);
```

Updating "Belongs To" Relationships

```
//updating a belongsTo relationship
$account = App\Account::find(10);
$user->account()->associate($account);            //set the foreign key on the child model
$user->save();

//removing a belongsTo relationship
$user->account()->dissociate();          //reset the foreign key on the child model
$user->save();
```

Many To Many Relationships

Attaching / Detaching

Attach a role to a user by inserting a record in the intermediate table that joins the models

```
$user = App\User::find(1);
$user->roles()->attach($roleId);

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


// Detach/remove a single role from the user...
$user->roles()->detach($roleId);

// Detach/remove all roles from the user...
$user->roles()->detach();
```

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]);
```

```
$user = App\User::find(1);
$user->roles()->updateExistingPivot($roleId, $attributes); //update an existing row in pivot table
```

Syncing For Convenience

only the IDs in the array will exist in the intermediate table:

```
$user->roles()->sync([1, 2, 3]);
//pass additional intermediate table values with the IDs
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

Touching Parent Timestamps::

update the parent's timestamp when the child model is updated

```php
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');
    }
}
```

when you update a `Comment`, the owning `Post` will have its `updated_at` column updated

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