VERSION

10.x

Eloquent: Relationships

Introduction

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 a variety of common relationships:

- One To One
- One To Many

- Many To Many
- Has One Through
- Has Many Through
- One To One (Polymorphic)
- One To Many (Polymorphic)
- Many To Many (Polymorphic)

Defining Relationships

Eloquent relationships are defined as methods on your Eloquent model classes. Since relationships also serve as powerful query builders, defining relationships as methods provides powerful method chaining and querying capabilities. For example, we may chain additional query constraints on this `posts` relationship:

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

But, before diving too deep into using relationships, let's learn how to define each type of relationship supported by Eloquent.

One To One

A one-to-one relationship is a very basic type of database relationship. For example, a `User` model might be associated with one `Phone` model. To define this relationship, we will place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result. The `hasOne` method is available to your model via the model's `Illuminate\Database\Eloquent\Model` base class:

```
<?php

namespace App\Models;
```

```php
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model
{
    /**
     * Get the phone associated with the user.
     */
    public function phone(): HasOne
    {
        return $this->hasOne(Phone::class);
    }
}
```

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

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

Eloquent determines the foreign key of the relationship based on the parent 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:

```php
return $this->hasOne(Phone::class, 'foreign_key');
```

Additionally, Eloquent assumes that the foreign key should have a value matching the primary key 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 primary key value other than `id` or your model's `$primaryKey` property, you may pass a third argument to the `hasOne` method:

```php
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

Defining The Inverse Of The Relationship

So, we can access the `Phone` model from our `User` model. Next, 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
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

When invoking the `user` method, Eloquent will attempt to find a `User` model that has an `id` which matches the `user_id` column on the `Phone` model.

Eloquent determines the foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. So, in this case, Eloquent assumes that the `Phone` model has a `user_id` column. 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:

```php
/**
 * Get the user that owns the phone.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key');
}
```

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

```php
/**
 * Get the user that owns the phone.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

One To Many

A one-to-many relationship is used to define relationships where a single model is the parent to one or more child 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 defining a method on your Eloquent model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model
```

```
{
    /**
     * Get the comments for the blog post.
     */
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

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

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

```
use App\Models\Post;

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

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

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the `comments` method and continuing to chain conditions onto the query:

```
$comment = 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(Comment::class, 'foreign_key');

return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

One To Many (Inverse) / Belongs To

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 method on the child model which calls the `belongsTo` method:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

Once the relationship has been defined, we can retrieve a comment's parent post by accessing the `post` "dynamic relationship property":

```php
use App\Models\Comment;

$comment = Comment::find(1);

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

In the example above, Eloquent will attempt to find a `Post` model that has an `id` which matches the `post_id` column on the `Comment` model.

Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with a `_` followed by the name of the parent model's primary key column. So, in this example, Eloquent will assume the `Post` model's foreign key on the `comments` table is `post_id`.

However, if the foreign key for your relationship does not follow these conventions, you may pass a custom foreign key name as the second argument to the `belongsTo` method:

```php
/**
 * Get the post that owns the comment.
 */
public function post(): BelongsTo
{
    return $this->belongsTo(Post::class, 'foreign_key');
}
```

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

```php
/**
 * Get the post that owns the comment.
 */
public function post(): BelongsTo
```

```
{
    return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

Default Models

The `belongsTo`, `hasOne`, `hasOneThrough`, and `morphOne` relationships allow you to define a default model that will be returned if the given relationship is `null`. This pattern is often referred to as the Null Object pattern and can help remove conditional checks in your code. In the following example, the `user` relation will return an empty `App\Models\User` model if no user is attached to the `Post` model:

```
/**
 * Get the author of the post.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault();
}
```

To populate the default model with attributes, you may pass an array or closure to the `withDefault` method:

```
/**
 * Get the author of the post.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Guest Author',
    ]);
}

/**
 * Get the author of the post.
 */
```

```php
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault(function (User $user, Post
        $user->name = 'Guest Author';
    });
}
```

Querying Belongs To Relationships

When querying for the children of a "belongs to" relationship, you may manually build the `where` clause to retrieve the corresponding Eloquent models:

```php
use App\Models\Post;

$posts = Post::where('user_id', $user->id)->get();
```

However, you may find it more convenient to use the `whereBelongsTo` method, which will automatically determine the proper relationship and foreign key for the given model:

```php
$posts = Post::whereBelongsTo($user)->get();
```

You may also provide a collection instance to the `whereBelongsTo` method. When doing so, Laravel will retrieve models that belong to any of the parent models within the collection:

```php
$users = User::where('vip', true)->get();

$posts = Post::whereBelongsTo($users)->get();
```

By default, Laravel will determine the relationship associated with the given model based on the class name of the model; however, you may specify the relationship

name manually by providing it as the second argument to the `whereBelongsTo` method:

```
$posts = Post::whereBelongsTo($user, 'author')->get();
```

Has One Of Many

Sometimes a model may have many related models, yet you want to easily retrieve the "latest" or "oldest" related model of the relationship. For example, a `User` model may be related to many `Order` models, but you want to define a convenient way to interact with the most recent order the user has placed. You may accomplish this using the `hasOne` relationship type combined with the `ofMany` methods:

```
/**
 * Get the user's most recent order.
 */
public function latestOrder(): HasOne
{
    return $this->hasOne(Order::class)->latestOfMany();
}
```

Likewise, you may define a method to retrieve the "oldest", or first, related model of a relationship:

```
/**
 * Get the user's oldest order.
 */
public function oldestOrder(): HasOne
{
    return $this->hasOne(Order::class)->oldestOfMany();
}
```

By default, the `latestOfMany` and `oldestOfMany` methods will retrieve the latest or oldest related model based on the model's primary key, which must be sortable. However, sometimes you may wish to retrieve a single model from a larger relationship using a different sorting criteria.

For example, using the `ofMany` method, you may retrieve the user's most expensive order. The `ofMany` method accepts the sortable column as its first argument and which aggregate function (`min` or `max`) to apply when querying for the related model:

```
/**
 * Get the user's largest order.
 */
public function largestOrder(): HasOne
{
    return $this->hasOne(Order::class)->ofMany('price', 'max');
}
```

Warning
Because PostgreSQL does not support executing the `MAX` function against UUID columns, it is not currently possible to use one-of-many relationships in combination with PostgreSQL UUID columns.

Converting "Many" Relationships To Has One Relationships

Often, when retrieving a single model using the `latestOfMany`, `oldestOfMany`, or `ofMany` methods, you already have a "has many" relationship defined for the same model. For convenience, Laravel allows you to easily convert this relationship into a "has one" relationship by invoking the `one` method on the relationship:

```
/**
 * Get the user's orders.
 */
public function orders(): HasMany
{
    return $this->hasMany(Order::class);
```

```
}

/**
 * Get the user's largest order.
 */
public function largestOrder(): HasOne
{
    return $this->orders()->one()->ofMany('price', 'max');
}
```

Advanced Has One Of Many Relationships

It is possible to construct more advanced "has one of many" relationships. For example, a `Product` model may have many associated `Price` models that are retained in the system even after new pricing is published. In addition, new pricing data for the product may be able to be published in advance to take effect at a future date via a `published_at` column.

So, in summary, we need to retrieve the latest published pricing where the published date is not in the future. In addition, if two prices have the same published date, we will prefer the price with the greatest ID. To accomplish this, we must pass an array to the `ofMany` method that contains the sortable columns which determine the latest price. In addition, a closure will be provided as the second argument to the `ofMany` method. This closure will be responsible for adding additional publish date constraints to the relationship query:

```
/**
 * Get the current pricing for the product.
 */
public function currentPricing(): HasOne
{
    return $this->hasOne(Price::class)->ofMany([
        'published_at' => 'max',
        'id' => 'max',
    ], function (Builder $query) {
        $query->where('published_at', '<', now());
```

```
    });
}
```

Has One Through

The "has-one-through" relationship defines a one-to-one relationship with another model. However, this relationship indicates that the declaring model can be matched with one instance of another model by proceeding *through* a third model.

For example, in a vehicle repair shop application, each `Mechanic` model may be associated with one `Car` model, and each `Car` model may be associated with one `Owner` model. While the mechanic and the owner have no direct relationship within the database, the mechanic can access the owner *through* the `Car` model. Let's look at the tables necessary to define this relationship:

```
mechanics
    id - integer
    name - string

cars
    id - integer
    model - string
    mechanic_id - integer

owners
    id - integer
    name - string
    car_id - integer
```

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

```
<?php
```

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOneThrough;

class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner(): HasOneThrough
    {
        return $this->hasOneThrough(Owner::class, Car::class);
    }
}
```

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

Or, if the relevant relationships have already been defined on all of the models involved in the relationship, you may fluently define a "has-one-through" relationship by invoking the `through` method and supplying the names of those relationships. For example, if the `Mechanic` model has a `cars` relationship and the `Car` model has an `owner` relationship, you may define a "has-one-through" relationship connecting the mechanic and the owner like so:

```php
// String based syntax...
return $this->through('cars')->has('owner');

// Dynamic syntax...
return $this->throughCars()->hasOwner();
```

Key Conventions

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 `hasOneThrough` 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. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```
class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner(): HasOneThrough
    {
        return $this->hasOneThrough(
            Owner::class,
            Car::class,
            'mechanic_id', // Foreign key on the cars table...
            'car_id', // Foreign key on the owners table...
            'id', // Local key on the mechanics table...
            'id' // Local key on the cars table...
        );
    }
}
```

Or, as discussed earlier, if the relevant relationships have already been defined on all of the models involved in the relationship, you may fluently define a "has-one-through" relationship by invoking the `through` method and supplying the names of those relationships. This approach offers the advantage of reusing the key conventions already defined on the existing relationships:

```
// String based syntax...
return $this->through('cars')->has('owner');

// Dynamic syntax...
```

```
return $this->throughCars()->hasOwner();
```

Has Many Through

The "has-many-through" relationship provides a convenient way to access distant relations via an intermediate relation. For example, let's assume we are building a deployment platform like Laravel Vapor. A `Project` model might access many `Deployment` models through an intermediate `Environment` model. Using this example, you could easily gather all deployments for a given project. Let's look at the tables required to define this relationship:

```
projects
    id - integer
    name - string

environments
    id - integer
    project_id - integer
    name - string

deployments
    id - integer
    environment_id - integer
    commit_hash - string
```

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

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasManyThrough;
```

```php
class Project extends Model
{
    /**
     * Get all of the deployments for the project.
     */
    public function deployments(): HasManyThrough
    {
        return $this->hasManyThrough(Deployment::class, Environment::class);
    }
}
```

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.

Or, if the relevant relationships have already been defined on all of the models involved in the relationship, you may fluently define a "has-many-through" relationship by invoking the `through` method and supplying the names of those relationships. For example, if the `Project` model has a `environments` relationship and the `Environment` model has a `deployments` relationship, you may define a "has-many-through" relationship connecting the project and the deployments like so:

```php
// String based syntax...
return $this->through('environments')->has('deployments');

// Dynamic syntax...
return $this->throughEnvironments()->hasDeployments();
```

Though the `Deployment` model's table does not contain a `project_id` column, the `hasManyThrough` relation provides access to a project's deployments via `$project->deployments`. To retrieve these models, Eloquent inspects the `project_id` column on the intermediate `Environment` model's table. After finding the relevant environment IDs, they are used to query the `Deployment` model's table.

Key Conventions

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. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```
class Project extends Model
{
    public function deployments(): HasManyThrough
    {
        return $this->hasManyThrough(
            Deployment::class,
            Environment::class,
            'project_id', // Foreign key on the environments table...
            'environment_id', // Foreign key on the deployments table...
            'id', // Local key on the projects table...
            'id' // Local key on the environments table...
        );
    }
}
```

Or, as discussed earlier, if the relevant relationships have already been defined on all of the models involved in the relationship, you may fluently define a "has-many-through" relationship by invoking the `through` method and supplying the names of those relationships. This approach offers the advantage of reusing the key conventions already defined on the existing relationships:

```
// String based syntax...
return $this->through('environments')->has('deployments');

// Dynamic syntax...
return $this->throughEnvironments()->hasDeployments();
```

Many To Many Relationships

Many-to-many relations are slightly more complicated than `hasOne` and `hasMany` relationships. An example of a many-to-many relationship is a user that has many roles and those roles are also shared by other users in the application. For example, a user may be assigned the role of "Author" and "Editor"; however, those roles may also be assigned to other users as well. So, a user has many roles and a role has many users.

Table Structure

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 `user_id` and `role_id` columns. This table is used as an intermediate table linking the users and roles.

Remember, since a role can belong to many users, we cannot simply place a `user_id` column on the `roles` table. This would mean that a role could only belong to a single user. In order to provide support for roles being assigned to multiple users, the `role_user` table is needed. We can summarize the relationship's table structure like so:

```
users
    id - integer
    name - string

roles
    id - integer
    name - string

role_user
    user_id - integer
    role_id - integer
```

Model Structure

Many-to-many relationships are defined by writing a method that returns the result of the `belongsToMany` method. The `belongsToMany` method is provided by the `Illuminate\Database\Eloquent\Model` base class that is used by all of your application's Eloquent models. For example, let's define a `roles` method on our `User` model. The first argument passed to this method is the name of the related model class:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles(): BelongsToMany
    {
        return $this->belongsToMany(Role::class);
    }
}
```

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

```php
use App\Models\User;

$user = User::find(1);

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

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the `roles` method and continuing to chain conditions onto the query:

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

To determine the table name of the relationship's intermediate 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(Role::class, 'role_user');
```

In addition to customizing the name of the intermediate 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(Role::class, 'role_user', 'user_id', 'role_id');
```

Defining The Inverse Of The Relationship

To define the "inverse" of a many-to-many relationship, you should define a method on the related model which also returns the result of the `belongsToMany` method. To complete our user / role example, let's define the `users` method on the `Role` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
```

```php
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class);
    }
}
```

As you can see, the relationship is defined exactly the same as its `User` model counterpart with the exception of referencing the `App\Models\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` model has many `Role` models that it is related to. After accessing this relationship, we may access the intermediate table using the `pivot` attribute on the models:

```php
use App\Models\User;

$user = 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.

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

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by');
```

If you would like your intermediate table to have `created_at` and `updated_at` timestamps that are automatically maintained by Eloquent, call the `withTimestamps` method when defining the relationship:

```
return $this->belongsToMany(Role::class)->withTimestamps();
```

Warning
Intermediate tables that utilize Eloquent's automatically maintained timestamps are required to have both `created_at` and `updated_at` timestamp columns.

Customizing The `pivot` Attribute Name

As noted previously, attributes from the intermediate table may be accessed on models via the `pivot` attribute. However, you are free to customize the name of this attribute to better reflect its purpose within your application.

For example, if your application contains users that may subscribe to podcasts, you likely have a many-to-many relationship between users and podcasts. If this is the case, you may wish to rename your intermediate table attribute to `subscription` instead of `pivot`. This can be done using the `as` method when defining the relationship:

```
return $this->belongsToMany(Podcast::class)
                ->as('subscription')
```

```
                ->withTimestamps();
```

Once the custom intermediate table attribute has been specified, you may access the intermediate table data using the customized name:

```
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

Filtering Queries Via Intermediate Table Columns

You can also filter the results returned by `belongsToMany` relationship queries using the `wherePivot`, `wherePivotIn`, `wherePivotNotIn`, `wherePivotBetween`, `wherePivotNotBetween`, `wherePivotNull`, and `wherePivotNotNull` methods when defining the relationship:

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

return $this->belongsToMany(Role::class)
                ->wherePivotIn('priority', [1, 2]);

return $this->belongsToMany(Role::class)
                ->wherePivotNotIn('priority', [1, 2]);

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotBetween('created_at', ['2020-01-01 00:00:00', '2020-

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotNotBetween('created_at', ['2020-01-01 00:00:00', '2(

return $this->belongsToMany(Podcast::class)
```

```
                ->as('subscriptions')
                ->wherePivotNull('expired_at');

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotNotNull('expired_at');
```

Ordering Queries Via Intermediate Table Columns

You can order the results returned by `belongsToMany` relationship queries using the `orderByPivot` method. In the following example, we will retrieve all of the latest badges for the user:

```
return $this->belongsToMany(Badge::class)
                ->where('rank', 'gold')
                ->orderByPivot('created_at', 'desc');
```

Defining Custom Intermediate Table Models

If you would like to define a custom model to represent the intermediate table of your many-to-many relationship, you may call the `using` method when defining the relationship. Custom pivot models give you the opportunity to define additional behavior on the pivot model, such as methods and casts.

Custom many-to-many pivot models should extend the `Illuminate\Database\Eloquent\Relations\Pivot` class while custom polymorphic many-to-many pivot models should extend the `Illuminate\Database\Eloquent\Relations\MorphPivot` class. For example, we may define a `Role` model which uses a custom `RoleUser` pivot model:

```
<?php

namespace App\Models;
```

```php
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class)->using(RoleUser::class);
    }
}
```

When defining the `RoleUser` model, you should extend the `Illuminate\Database\Eloquent\Relations\Pivot` class:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot
{
    // ...
}
```

Warning

Pivot models may not use the `SoftDeletes` trait. If you need to soft delete pivot records consider converting your pivot model to an actual Eloquent model.

Custom Pivot Models And Incrementing IDs

If you have defined a many-to-many relationship that uses a custom pivot model, and that pivot model has an auto-incrementing primary key, you should ensure

your custom pivot model class defines an `incrementing` property that is set to `true`.

```
/**
 * Indicates if the IDs are auto-incrementing.
 *
 * @var bool
 */
public $incrementing = true;
```

Polymorphic Relationships

A polymorphic relationship allows the child model to belong to more than one type of model using a single association. For example, imagine you are building an application that allows users to share blog posts and videos. In such an application, a `Comment` model might belong to both the `Post` and `Video` models.

One To One (Polymorphic)

Table Structure

A one-to-one polymorphic relation is similar to a typical one-to-one relation; however, the child model can belong to more than one type of model using a single association. For example, a blog `Post` and a `User` may share a polymorphic relation to an `Image` model. Using a one-to-one polymorphic relation allows you to have a single table of unique images that may be associated with posts and users. First, let's examine the table structure:

```
posts
    id - integer
    name - string

users
    id - integer
    name - string
```

```
images
    id - integer
    url - string
    imageable_id - integer
    imageable_type - string
```

Note the `imageable_id` and `imageable_type` columns on the `images` table. The `imageable_id` column will contain the ID value of the post or user, while the `imageable_type` column will contain the class name of the parent model. The `imageable_type` column is used by Eloquent to determine which "type" of parent model to return when accessing the `imageable` relation. In this case, the column would contain either `App\Models\Post` or `App\Models\User`.

Model Structure

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

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class Image extends Model
{
    /**
     * Get the parent imageable model (user or post).
     */
    public function imageable(): MorphTo
    {
        return $this->morphTo();
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;
```

```php
class Post extends Model
{
    /**
     * Get the post's image.
     */
    public function image(): MorphOne
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;

class User extends Model
{
    /**
     * Get the user's image.
     */
    public function image(): MorphOne
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your models. For example, to retrieve the image for a post, we can access the `image` dynamic relationship property:

```php
use App\Models\Post;

$post = Post::find(1);

$image = $post->image;
```

You may retrieve the parent of the polymorphic model by accessing the name of the method that performs the call to `morphTo`. In this case, that is the `imageable` method on the `Image` model. So, we will access that method as a dynamic relationship property:

```
use App\Models\Image;

$image = Image::find(1);

$imageable = $image->imageable;
```

The `imageable` relation on the `Image` model will return either a `Post` or `User` instance, depending on which type of model owns the image.

Key Conventions

If necessary, you may specify the name of the "id" and "type" columns utilized by your polymorphic child model. If you do so, ensure that you always pass the name of the relationship as the first argument to the `morphTo` method. Typically, this value should match the method name, so you may use PHP's `__FUNCTION__` constant:

```
/**
 * Get the model that the image belongs to.
 */
public function imageable(): MorphTo
{
    return $this->morphTo(__FUNCTION__, 'imageable_type', 'imageable_id');
}
```

One To Many (Polymorphic)

Table Structure

A one-to-many polymorphic relation is similar to a typical one-to-many relation; however, the child model can belong to more than one type of model using a

single association. For example, imagine users of your application can "comment" on posts and videos. Using polymorphic relationships, you may use a single `comments` table to contain comments for both posts and videos. First, let's examine the table structure required to build this relationship:

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

videos
    id - integer
    title - string
    url - string

comments
    id - integer
    body - text
    commentable_id - integer
    commentable_type - string
```

Model Structure

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

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class Comment extends Model
{
    /**
     * Get the parent commentable model (post or video).
     */
    public function commentable(): MorphTo
```

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

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your model's dynamic relationship properties. For example, to access all of the comments for a post, we can use the `comments` dynamic property:

```php
use App\Models\Post;

$post = Post::find(1);

foreach ($post->comments as $comment) {
    // ...
}
```

You may also retrieve the parent of a polymorphic child model by accessing the name of the method that performs the call to `morphTo`. In this case, that is the `commentable` method on the `Comment` model. So, we will access that method as a dynamic relationship property in order to access the comment's parent model:

```php
use App\Models\Comment;

$comment = Comment::find(1);

$commentable = $comment->commentable;
```

The `commentable` relation on the `Comment` model will return either a `Post` or `Video` instance, depending on which type of model is the comment's parent.

One Of Many (Polymorphic)

Sometimes a model may have many related models, yet you want to easily retrieve the "latest" or "oldest" related model of the relationship. For example, a `User` model may be related to many `Image` models, but you want to define a convenient way to interact with the most recent image the user has uploaded. You may accomplish this using the `morphOne` relationship type combined with the `ofMany` methods:

```php
/**
 * Get the user's most recent image.
 */
```

```php
public function latestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->latestOfMany();
}
```

Likewise, you may define a method to retrieve the "oldest", or first, related model of a relationship:

```php
/**
 * Get the user's oldest image.
 */
public function oldestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->oldestOfMany();
}
```

By default, the `latestOfMany` and `oldestOfMany` methods will retrieve the latest or oldest related model based on the model's primary key, which must be sortable. However, sometimes you may wish to retrieve a single model from a larger relationship using a different sorting criteria.

For example, using the `ofMany` method, you may retrieve the user's most "liked" image. The `ofMany` method accepts the sortable column as its first argument and which aggregate function (`min` or `max`) to apply when querying for the related model:

```php
/**
 * Get the user's most popular image.
 */
public function bestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->ofMany('likes', 'max');
}
```

Note

It is possible to construct more advanced "one of many" relationships. For more information, please consult the has one of many documentation.

Many To Many (Polymorphic)

Table Structure

Many-to-many polymorphic relations are slightly more complicated than "morph one" and "morph many" relationships. For example, a `Post` model and `Video` model could share a polymorphic relation to a `Tag` model. Using a many-to-many polymorphic relation in this situation would allow your application to have a single table of unique tags that may be associated with posts or videos. First, let's examine the table structure required to build this relationship:

```
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
```

Note

Before diving into polymorphic many-to-many relationships, you may benefit from reading the documentation on typical many-to-many relationships.

Model Structure

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

The `morphToMany` method accepts the name of the related model as well as the "relationship name". Based on the name we assigned to our intermediate table name and the keys it contains, we will refer to the relationship as "taggable":

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphToMany;

class Post extends Model
{
    /**
     * Get all of the tags for the post.
     */
    public function tags(): MorphToMany
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

Defining The Inverse Of The Relationship

Next, on the `Tag` model, you should define a method for each of its possible parent models. So, in this example, we will define a `posts` method and a `videos` method. Both of these methods should return the result of the `morphedByMany` method.

The `morphedByMany` method accepts the name of the related model as well as the "relationship name". Based on the name we assigned to our intermediate table name and the keys it contains, we will refer to the relationship as "taggable":

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphToMany;

class Tag extends Model
{
    /**
     * Get all of the posts that are assigned this tag.
     */
    public function posts(): MorphToMany
    {
        return $this->morphedByMany(Post::class, 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos(): MorphToMany
    {
        return $this->morphedByMany(Video::class, '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 may use the `tags` dynamic relationship property:

```php
use App\Models\Post;

$post = Post::find(1);

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

```
}
```

You may retrieve the parent of a polymorphic relation from the polymorphic child model by accessing the name of the method that performs the call to `morphedByMany`. In this case, that is the `posts` or `videos` methods on the `Tag` model:

```
use App\Models\Tag;

$tag = Tag::find(1);

foreach ($tag->posts as $post) {
    // ...
}

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

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 one-to-many relationship example above where a `Comment` model may belong to a `Post` or a `Video` model, the default `commentable_type` would be either `App\Models\Post` or `App\Models\Video`, respectively. However, you may wish to decouple these values from your application's internal structure.

For example, instead of using the model names as the "type", we may use simple strings such as `post` and `video`. By doing so, the polymorphic "type" column values in our database will remain valid even if the models are renamed:

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::enforceMorphMap([
    'post' => 'App\Models\Post',
```

```
    'video' => 'App\Models\Video',
]);
```

You may call the `enforceMorphMap` method in the `boot` method of your `App\Providers\AppServiceProvider` class or create a separate service provider if you wish.

You may determine the morph alias of a given model at runtime using the model's `getMorphClass` method. Conversely, you may determine the fully-qualified class name associated with a morph alias using the `Relation::getMorphedModel` method:

```
use Illuminate\Database\Eloquent\Relations\Relation;

$alias = $post->getMorphClass();

$class = Relation::getMorphedModel($alias);
```

Warning
When adding a "morph map" to your existing application, every morphable `*_type` column value in your database that still contains a fully-qualified class will need to be converted to its "map" name.

Dynamic Relationships

You may use the `resolveRelationUsing` method to define relations between Eloquent models at runtime. While not typically recommended for normal application development, this may occasionally be useful when developing Laravel packages.

The `resolveRelationUsing` method accepts the desired relationship name as its first argument. The second argument passed to the method should be a closure that accepts the model instance and returns a valid Eloquent relationship definition. Typically, you should configure dynamic relationships within the boot method of a service provider:

```php
use App\Models\Order;
use App\Models\Customer;

Order::resolveRelationUsing('customer', function (Order $orderModel) {
    return $orderModel->belongsTo(Customer::class, 'customer_id');
});
```

Warning

When defining dynamic relationships, always provide explicit key name arguments to the Eloquent relationship methods.

Querying Relations

Since all Eloquent relationships are defined via methods, you may call those methods to obtain an instance of the relationship without actually executing a query to load the related models. In addition, all types of Eloquent relationships also serve as query builders, allowing you to continue to chain constraints onto the relationship query before finally executing the SQL query against your database.

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

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts(): HasMany
```

```
    {
        return $this->hasMany(Post::class);
    }
}
```

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

```
use App\Models\User;

$user = User::find(1);

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

You are able to use any of the Laravel query builder's methods on the relationship, so be sure to explore the query builder documentation to learn about all of the methods that are available to you.

Chaining `orWhere` Clauses After Relationships

As demonstrated in the example above, you are free to add additional constraints to relationships when querying them. However, use caution when chaining `orWhere` clauses onto a relationship, as the `orWhere` clauses will be logically grouped at the same level as the relationship constraint:

```
$user->posts()
        ->where('active', 1)
        ->orWhere('votes', '>=', 100)
        ->get();
```

The example above will generate the following SQL. As you can see, the `or` clause instructs the query to return *any* post with greater than 100 votes. The query is no longer constrained to a specific user:

```
select *
from posts
where user_id = ? and active = 1 or votes >= 100
```

In most situations, you should use logical groups to group the conditional checks between parentheses:

```
use Illuminate\Database\Eloquent\Builder;

$user->posts()
        ->where(function (Builder $query) {
            return $query->where('active', 1)
                         ->orWhere('votes', '>=', 100);
        })
        ->get();
```

The example above will produce the following SQL. Note that the logical grouping has properly grouped the constraints and the query remains constrained to a specific user:

```
select *
from posts
where user_id = ? and (active = 1 or votes >= 100)
```

Relationship Methods Vs. Dynamic Properties

If you do not need to add additional constraints to an Eloquent relationship query, you may 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:

```
use App\Models\User;

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

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

Dynamic relationship properties perform "lazy loading", meaning they will only load their relationship data when you actually access them. Because of this, developers often use 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 retrieving model records, 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` and `orHas` methods:

```
use App\Models\Post;

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

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

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

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

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

If you need even more power, you may use the `whereHas` and `orWhereHas` methods to define additional query constraints on your `has` queries, such as inspecting the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

// Retrieve posts with at least one comment containing words like code%...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();

// Retrieve posts with at least ten comments containing words like code%...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
}, '>=', 10)->get();
```

Warning

Eloquent does not currently support querying for relationship existence across databases. The relationships must exist within the same database.

Inline Relationship Existence Queries

If you would like to query for a relationship's existence with a single, simple where condition attached to the relationship query, you may find it more convenient to use the `whereRelation`, `orWhereRelation`, `whereMorphRelation`, and `orWhereMorphRelation` methods. For example, we may query for all posts that have unapproved comments:

```
use App\Models\Post;

$posts = Post::whereRelation('comments', 'is_approved', false)->get();
```

Of course, like calls to the query builder's `where` method, you may also specify an operator:

```php
$posts = Post::whereRelation(
    'comments', 'created_at', '>=', now()->subHour()
)->get();
```

Querying Relationship Absence

When retrieving model records, you may wish to limit your results based on the absence of a relationship. For example, imagine you want to retrieve all blog posts that **don't** have any comments. To do so, you may pass the name of the relationship to the `doesntHave` and `orDoesntHave` methods:

```php
use App\Models\Post;

$posts = Post::doesntHave('comments')->get();
```

If you need even more power, you may use the `whereDoesntHave` and `orWhereDoesntHave` methods to add additional query constraints to your `doesntHave` queries, such as inspecting the content of a comment:

```php
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();
```

You may use "dot" notation to execute a query against a nested relationship. For example, the following query will retrieve all posts that do not have comments; however, posts that have comments from authors that are not banned will be included in the results:

```php
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments.author', function (Builder $query) {
```

```
    $query->where('banned', 0);
})->get();
```

Querying Morph To Relationships

To query the existence of "morph to" relationships, you may use the `whereHasMorph` and `whereDoesntHaveMorph` methods. These methods accept the name of the relationship as their first argument. Next, the methods accept the names of the related models that you wish to include in the query. Finally, you may provide a closure which customizes the relationship query:

```
use App\Models\Comment;
use App\Models\Post;
use App\Models\Video;
use Illuminate\Database\Eloquent\Builder;

// Retrieve comments associated to posts or videos with a title like code%...
$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();

// Retrieve comments associated to posts with a title not like code%...
$comments = Comment::whereDoesntHaveMorph(
    'commentable',
    Post::class,
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();
```

You may occasionally need to add query constraints based on the "type" of the related polymorphic model. The closure passed to the `whereHasMorph` method may

receive a `$type` value as its second argument. This argument allows you to inspect the "type" of the query that is being built:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query, string $type) {
        $column = $type === Post::class ? 'content' : 'title';

        $query->where($column, 'like', 'code%');
    }
)->get();
```

Querying All Related Models

Instead of passing an array of possible polymorphic models, you may provide `*` as a wildcard value. This will instruct Laravel to retrieve all of the possible polymorphic types from the database. Laravel will execute an additional query in order to perform this operation:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph('commentable', '*', function (Builder $query)
    $query->where('title', 'like', 'foo%');
})->get();
```

Aggregating Related Models

Counting Related Models

Sometimes you may want to count the number of related models for a given relationship without actually loading the models. To accomplish this, you may use

the `withCount` method. The `withCount` method will place a `{relation}_count` attribute on the resulting models:

```
use App\Models\Post;

$posts = Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

By passing an array to the `withCount` method, you may add the "counts" for multiple relations as well as add additional constraints to the queries:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount(['votes', 'comments' => function (Builder $query) {
    $query->where('content', 'like', 'code%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

You may also alias the relationship count result, allowing multiple counts on the same relationship:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount([
    'comments',
    'comments as pending_comments_count' => function (Builder $query) {
        $query->where('approved', false);
    },
])->get();

echo $posts[0]->comments_count;
```

```php
echo $posts[0]->pending_comments_count;
```

Deferred Count Loading

Using the `loadCount` method, you may load a relationship count after the parent model has already been retrieved:

```php
$book = Book::first();

$book->loadCount('genres');
```

If you need to set additional query constraints on the count query, you may pass an array keyed by the relationships you wish to count. The array values should be closures which receive the query builder instance:

```php
$book->loadCount(['reviews' => function (Builder $query) {
    $query->where('rating', 5);
}])
```

Relationship Counting & Custom Select Statements

If you're combining `withCount` with a `select` statement, ensure that you call `withCount` after the `select` method:

```php
$posts = Post::select(['title', 'body'])
                ->withCount('comments')
                ->get();
```

Other Aggregate Functions

In addition to the `withCount` method, Eloquent provides `withMin`, `withMax`, `withAvg`, `withSum`, and `withExists` methods. These methods will place a `{relation}_{function}_{column}` attribute on your resulting models:

```php
use App\Models\Post;

$posts = Post::withSum('comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->comments_sum_votes;
}
```

If you wish to access the result of the aggregate function using another name, you may specify your own alias:

```php
$posts = Post::withSum('comments as total_comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->total_comments;
}
```

Like the `loadCount` method, deferred versions of these methods are also available. These additional aggregate operations may be performed on Eloquent models that have already been retrieved:

```php
$post = Post::first();

$post->loadSum('comments', 'votes');
```

If you're combining these aggregate methods with a `select` statement, ensure that you call the aggregate methods after the `select` method:

```php
$posts = Post::select(['title', 'body'])
                ->withExists('comments')
                ->get();
```

Counting Related Models On Morph To Relationships

If you would like to eager load a "morph to" relationship, as well as related model counts for the various entities that may be returned by that relationship, you may utilize the `with` method in combination with the `morphTo` relationship's `morphWithCount` method.

In this example, let's assume that `Photo` and `Post` models may create `ActivityFeed` models. We will assume the `ActivityFeed` model defines a "morph to" relationship named `parentable` that allows us to retrieve the parent `Photo` or `Post` model for a given `ActivityFeed` instance. Additionally, let's assume that `Photo` models "have many" `Tag` models and `Post` models "have many" `Comment` models.

Now, let's imagine we want to retrieve `ActivityFeed` instances and eager load the `parentable` parent models for each `ActivityFeed` instance. In addition, we want to retrieve the number of tags that are associated with each parent photo and the number of comments that are associated with each parent post:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::with([
    'parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWithCount([
            Photo::class => ['tags'],
            Post::class => ['comments'],
        ]);
    }])->get();
```

Deferred Count Loading

Let's assume we have already retrieved a set of `ActivityFeed` models and now we would like to load the nested relationship counts for the various `parentable` models associated with the activity feeds. You may use the `loadMorphCount` method to accomplish this:

```
$activities = ActivityFeed::with('parentable')->get();
```

```php
$activities->loadMorphCount('parentable', [
    Photo::class => ['tags'],
    Post::class => ['comments'],
]);
```

Eager Loading

When accessing Eloquent relationships as properties, the related models are "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 "belongs to" to an `Author` model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author(): BelongsTo
    {
        return $this->belongsTo(Author::class);
    }
}
```

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

```php
use App\Models\Book;
```

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

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

This loop will execute one query to retrieve all of the books within the database table, then another query for each book in order to retrieve the book's author. So, if we have 25 books, the code above would run 26 queries: one 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 two queries. When building a query, you may specify which relationships should be eager loaded using the `with` method:

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

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

For this operation, only two queries will be executed - one query to retrieve all of the books and one query to retrieve all of the authors for all of the books:

```sql
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. To do so, just pass an array of relationships to the `with` method:

```php
$books = Book::with(['author', 'publisher'])->get();
```

Nested Eager Loading

To eager load a relationship's 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:

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

Alternatively, you may specify nested eager loaded relationships by providing a nested array to the `with` method, which can be convenient when eager loading multiple nested relationships:

```php
$books = Book::with([
    'author' => [
        'contacts',
        'publisher',
    ],
])->get();
```

Nested Eager Loading `morphTo` Relationships

If you would like to eager load a `morphTo` relationship, as well as nested relationships on the various entities that may be returned by that relationship, you may use the `with` method in combination with the `morphTo` relationship's `morphWith` method. To help illustrate this method, let's consider the following model:

```php
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class ActivityFeed extends Model
{
    /**
```

```
     * Get the parent of the activity feed record.
     */
    public function parentable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

In this example, let's assume `Event`, `Photo`, and `Post` models may create `ActivityFeed` models. Additionally, let's assume that `Event` models belong to a `Calendar` model, `Photo` models are associated with `Tag` models, and `Post` models belong to an `Author` model.

Using these model definitions and relationships, we may retrieve `ActivityFeed` model instances and eager load all `parentable` models and their respective nested relationships:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::query()
    ->with(['parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWith([
            Event::class => ['calendar'],
            Photo::class => ['tags'],
            Post::class => ['author'],
        ]);
    }])->get();
```

Eager Loading Specific Columns

You may not always need every column from the relationships you are retrieving. For this reason, Eloquent allows you to specify which columns of the relationship you would like to retrieve:

```
$books = Book::with('author:id,name,book_id')->get();
```

Warning

When using this feature, you should always include the `id` column and any relevant foreign key columns in the list of columns you wish to retrieve.

Eager Loading By Default

Sometimes you might want to always load some relationships when retrieving a model. To accomplish this, you may define a `$with` property on the model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Book extends Model
{
    /**
     * The relationships that should always be loaded.
     *
     * @var array
     */
    protected $with = ['author'];

    /**
     * Get the author that wrote the book.
     */
    public function author(): BelongsTo
    {
        return $this->belongsTo(Author::class);
    }

    /**
     * Get the genre of the book.
     */
    public function genre(): BelongsTo
    {
        return $this->belongsTo(Genre::class);
```

```
    }
}
```

If you would like to remove an item from the `$with` property for a single query, you may use the `without` method:

```
$books = Book::without('author')->get();
```

If you would like to override all items within the `$with` property for a single query, you may use the `withOnly` method:

```
$books = Book::withOnly('genre')->get();
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship but also specify additional query conditions for the eager loading query. You can accomplish this by passing an array of relationships to the `with` method where the array key is a relationship name and the array value is a closure that adds additional constraints to the eager loading query:

```
use App\Models\User;
use Illuminate\Contracts\Database\Eloquent\Builder;

$users = User::with(['posts' => function (Builder $query) {
    $query->where('title', 'like', '%code%');
}])->get();
```

In this example, Eloquent will only eager load posts where the post's `title` column contains the word `code`. You may call other query builder methods to further customize the eager loading operation:

```php
$users = User::with(['posts' => function (Builder $query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

Warning

The `limit` and `take` query builder methods may not be used when constraining eager loads.

Constraining Eager Loading Of `morphTo` Relationships

If you are eager loading a `morphTo` relationship, Eloquent will run multiple queries to fetch each type of related model. You may add additional constraints to each of these queries using the `MorphTo` relation's `constrain` method:

```php
use Illuminate\Database\Eloquent\Relations\MorphTo;

$comments = Comment::with(['commentable' => function (MorphTo $morphTo) {
    $morphTo->constrain([
        Post::class => function ($query) {
            $query->whereNull('hidden_at');
        },
        Video::class => function ($query) {
            $query->where('type', 'educational');
        },
    ]);
}])->get();
```

In this example, Eloquent will only eager load posts that have not been hidden and videos that have a `type` value of "educational".

Constraining Eager Loads With Relationship Existence

You may sometimes find yourself needing to check for the existence of a relationship while simultaneously loading the relationship based on the same conditions. For example, you may wish to only retrieve `User` models that have child

`Post` models matching a given query condition while also eager loading the matching posts. You may accomplish this using the `withWhereHas` method:

```
use App\Models\User;

$users = User::withWhereHas('posts', function ($query) {
    $query->where('featured', true);
})->get();
```

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:

```
use App\Models\Book;

$books = Book::all();

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

If you need to set additional query constraints on the eager loading query, you may pass an array keyed by the relationships you wish to load. The array values should be closure instances which receive the query instance:

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

To load a relationship only when it has not already been loaded, use the `loadMissing` method:

```php
$book->loadMissing('author');
```

Nested Lazy Eager Loading & `morphTo`

If you would like to eager load a `morphTo` relationship, as well as nested relationships on the various entities that may be returned by that relationship, you may use the `loadMorph` method.

This method accepts the name of the `morphTo` relationship as its first argument, and an array of model / relationship pairs as its second argument. To help illustrate this method, let's consider the following model:

```php
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class ActivityFeed extends Model
{
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

In this example, let's assume `Event`, `Photo`, and `Post` models may create `ActivityFeed` models. Additionally, let's assume that `Event` models belong to a `Calendar` model, `Photo` models are associated with `Tag` models, and `Post` models belong to an `Author` model.

Using these model definitions and relationships, we may retrieve `ActivityFeed` model instances and eager load all `parentable` models and their respective nested

relationships:

```
$activities = ActivityFeed::with('parentable')
    ->get()
    ->loadMorph('parentable', [
        Event::class => ['calendar'],
        Photo::class => ['tags'],
        Post::class => ['author'],
    ]);
```

Preventing Lazy Loading

As previously discussed, eager loading relationships can often provide significant performance benefits to your application. Therefore, if you would like, you may instruct Laravel to always prevent the lazy loading of relationships. To accomplish this, you may invoke the `preventLazyLoading` method offered by the base Eloquent model class. Typically, you should call this method within the `boot` method of your application's `AppServiceProvider` class.

The `preventLazyLoading` method accepts an optional boolean argument that indicates if lazy loading should be prevented. For example, you may wish to only disable lazy loading in non-production environments so that your production environment will continue to function normally even if a lazy loaded relationship is accidentally present in production code:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Model::preventLazyLoading(! $this->app->isProduction());
}
```

After preventing lazy loading, Eloquent will throw a `Illuminate\Database\LazyLoadingViolationException` exception when your application attempts to lazy load any Eloquent relationship.

You may customize the behavior of lazy loading violations using the `handleLazyLoadingViolationsUsing` method. For example, using this method, you may instruct lazy loading violations to only be logged instead of interrupting the application's execution with exceptions:

```
Model::handleLazyLoadingViolationUsing(function (Model $model, string $relation
    $class = get_class($model);

    info("Attempted to lazy load [{$relation}] on model [{$class}].");
});
```

Inserting & Updating Related Models

The `save` Method

Eloquent provides convenient methods for adding new models to relationships. For example, perhaps you need to add a new comment to a post. Instead of manually setting the `post_id` attribute on the `Comment` model you may insert the comment using the relationship's `save` method:

```
use App\Models\Comment;
use App\Models\Post;

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

$post = Post::find(1);

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

Note 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 = Post::find(1);

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

The `save` and `saveMany` methods will persist the given model instances, but will not add the newly persisted models to any in-memory relationships that are already loaded onto the parent model. If you plan on accessing the relationship after using the `save` or `saveMany` methods, you may wish to use the `refresh` method to reload the model and its relationships:

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

$post->refresh();

// All comments, including the newly saved comment...
$post->comments;
```

Recursively Saving Models & Relationships

If you would like to `save` your model and all of its associated relationships, you may use the `push` method. In this example, the `Post` model will be saved as well as its comments and the comment's authors:

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

```
$post->comments[0]->message = 'Message';
$post->comments[0]->author->name = 'Author Name';

$post->push();
```

The `pushQuietly` method may be used to save a model and its associated relationships without raising any events:

```
$post->pushQuietly();
```

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. The difference between `save` and `create` is that `save` accepts a full Eloquent model instance while `create` accepts a plain PHP `array`. The newly created model will be returned by the `create` method:

```
use App\Models\Post;

$post = Post::find(1);

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

You may use the `createMany` method to create multiple related models:

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

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

```
]);
```

The `createQuietly` and `createManyQuietly` methods may be used to create a model(s) without dispatching any events:

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

$user->posts()->createQuietly([
    'title' => 'Post title.',
]);

$user->posts()->createManyQuietly([
    ['title' => 'First post.'],
    ['title' => 'Second post.'],
]);
```

You may also use the `findOrNew`, `firstOrNew`, `firstOrCreate`, and `updateOrCreate` methods to create and update models on relationships.

Note
Before using the `create` method, be sure to review the mass assignment documentation.

Belongs To Relationships

If you would like to assign a child model to a new parent model, you may use the `associate` method. In this example, the `User` model defines a `belongsTo` relationship to the `Account` model. This `associate` method will set the foreign key on the child model:

```
use App\Models\Account;

$account = Account::find(10);

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

```
$user->save();
```

To remove a parent model from a child model, you may use the `dissociate` method. This method will set the relationship's foreign key to `null`:

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

$user->save();
```

Many To Many Relationships

Attaching / Detaching

Eloquent also provides methods to make working with many-to-many relationships more convenient. For example, let's imagine a user can have many roles and a role can have many users. You may use the `attach` method to attach a role to a user by inserting a record in the relationship's intermediate table:

```
use App\Models\User;

$user = 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]);
```

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

delete 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 = User::find(1);

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

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

Syncing Associations

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 given 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]);
```

If you would like to insert the same intermediate table values with each of the synced model IDs, you may use the `syncWithPivotValues` method:

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

If you do not want to detach existing IDs that are missing from the given array, you may use the `syncWithoutDetaching` method:

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

Toggling Associations

The many-to-many relationship also provides a `toggle` method which "toggles" the attachment status of the given related model IDs. If the given ID is currently attached, it will be detached. Likewise, if it is currently detached, it will be attached:

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

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

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

Updating A Record On The Intermediate Table

If you need to update an existing row in your relationship's intermediate table, you may use the `updateExistingPivot` method. This method accepts the intermediate record foreign key and an array of attributes to update:

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

```php
$user->roles()->updateExistingPivot($roleId, [
    'active' => false,
]);
```

Touching Parent Timestamps

When a model defines a `belongsTo` or `belongsToMany` relationship to 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.

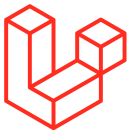
For example, when a `Comment` model is updated, you may want to automatically "touch" the `updated_at` timestamp of the owning `Post` so that it is set to the current date and time. To accomplish this, you may add a `touches` property to your child model containing the names of the relationships that should have their `updated_at` timestamps updated when the child model is updated:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * All of the relationships to be touched.
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Get the post that the comment belongs to.
     */
```

```php
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

Warning

Parent model timestamps will only be updated if the child model is updated using Eloquent's `save` method.



Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.

PARTNERS

WebReinvent

Vehikl

Tighten

64 Robots

Active Logic

Byte 5

Curotec

Cyber-Duck

DevSquad

Jump24

Kirschbaum

ECOSYSTEM

Breeze

Cashier

Dusk

Echo

Envoyer

Forge

Herd

Horizon

Inertia

Jetstream

Livewire

Nova

Octane

Pennant

Pint

Prompts

Sail

Sanctum

Scout

Socialite

Spark

Telescope

Vapor

Code highlighting provided by Torchlight