

# Eloquent: Relationships

- Introduction
- Defining Relationships
  - One To One
  - One To Many
  - One To Many (Inverse)
  - Many To Many
  - Has Many Through
  - Polymorphic Relations
  - Many To Many Polymorphic Relations
- Querying Relations
  - Relationship Methods Vs. Dynamic Properties
  - Querying Relationship Existence
  - Querying Relationship Absence
  - Counting Related Models
- Eager Loading
  - Constraining Eager Loads
  - Lazy Eager Loading
- Inserting & Updating Related Models
  - The `save` Method
  - The `create` Method
  - Belongs To Relationships
  - Many To Many Relationships
- Touching Parent Timestamps

## 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 several different types of relationships:

- One To One
- One To Many
- Many To Many
- Has Many Through
- Polymorphic Relations
- Many To Many Polymorphic Relations

## Defining Relationships

Eloquent relationships are defined as methods on your Eloquent model classes. Since, like Eloquent models themselves, relationships also serve as powerful **query builders**, defining relationships as

methods provides powerful method chaining and querying capabilities. For example, we may chain additional 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.

## One To One

A one-to-one relationship is a very basic relation. For example, a `User` model might be associated with one `Phone`. To define this relationship, we place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

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

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

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

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

Additionally, Eloquent assumes that the foreign key should have a value matching the `id` (or the custom `$primaryKey`) column of the parent. In other words, Eloquent will look for the value of the

user's `id` column in the `user_id` column of the `Phone` record. If you would like the relationship to use a value other than `id`, you may pass a third argument to the `hasOne` method specifying your custom key:

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

## Defining The Inverse Of The Relationship

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

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

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

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

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

```

/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}

```

## Default Models

The `belongsTo` relationship allows 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\User` model if no `user` is attached to the post:

```

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

/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo('App\User')->withDefault(function ($user) {
        $user->name = 'Guest Author';
    });
}

```

## One To Many

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

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

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

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

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

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

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

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

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

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

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

## One To Many (Inverse)

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

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

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

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

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

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

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

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

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');
}
```

## Many To Many

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

Many-to-many relationships are defined by writing a method that returns the result of the `belongsToMany` method. For example, let's define the `roles` method on our `User` model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

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

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

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

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

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

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

```
return $this->belongsToMany('App\Role', '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 to the `belongsToMany` method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to:

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

## Defining The Inverse Of The Relationship

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

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

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



## Retrieving Intermediate Table Columns

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

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

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

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

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

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

If you want your pivot table to have automatically maintained `created_at` and `updated_at` timestamps, use the `withTimestamps` method on the relationship definition:

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

## Customizing The `pivot` Attribute Name

As noted earlier, attributes from the intermediate table may be accessed on models using 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 probably have a many-to-many relationship between users and podcasts. If this is the case, you may wish to rename your intermediate table accessor to `subscription` instead of `pivot`. This can be done using the `as` method when defining the relationship:

```
return $this->belongsToMany('App\Podcast')
    ->as('subscription')
    ->withTimestamps();
```

Once this is done, 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 Relationships Via Intermediate Table Columns

You can also 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('priority', [1, 2]);
```

## Defining Custom Intermediate Table Models

If you would like to define a custom model to represent the intermediate table of your relationship, you may call the `using` method when defining the relationship. All custom models used to represent intermediate tables of relationships must extend the `Illuminate\Database\Eloquent\Relations\Pivot` class. For example, we may define a `Role` which uses a custom `UserRole` pivot model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

When defining the `UserRole` model, we will extend the `Pivot` class:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Relations\Pivot;

class UserRole extends Pivot
```

```
{  
    //  
}
```

## Has Many Through

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

```
countries  
  id - integer  
  name - string  
  
users  
  id - integer  
  country_id - integer  
  name - string  
  
posts  
  id - integer  
  user_id - integer  
  title - string
```

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

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

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Country extends Model  
{  
    /**  
     * Get all of the posts for the country.  
     */  
    public function posts()  
    {  
        return $this->hasManyThrough('App\Post', 'App\User');  
    }  
}
```

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

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasManyThrough` method. The third argument is the name of the foreign key on the intermediate model. 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 Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(
            'App\Post',
            'App\User',
            'country_id', // Foreign key on users table...
            'user_id', // Foreign key on posts table...
            'id', // Local key on countries table...
            'id' // Local key on users table...
        );
    }
}
```

## Polymorphic Relations

### Table Structure

Polymorphic relations allow a model to belong to more than one other model on a single association. For example, imagine users of your application can "comment" both posts and videos. Using polymorphic relationships, you can use a single `comments` table for both of these scenarios. 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
```

Two important columns to note are the `commentable_id` and `commentable_type` columns on the `comments` table. The `commentable_id` column will contain the ID value of the post or video, while the `commentable_type` column will contain the class name of the owning model. The `commentable_type` column is how the ORM determines which "type" of owning model to return when accessing the `commentable` relation.

## Model Structure

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

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get all of the owning commentable models.
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

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

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

## Retrieving Polymorphic Relations

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

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

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

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

```
$comment = App\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 owns the comment.

## Custom Polymorphic Types

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

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

Relation::morphMap([
    'posts' => 'App\Post',
    'videos' => 'App\Video',
]);
```

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

### Table Structure

In addition to traditional polymorphic relations, you may also define "many-to-many" polymorphic relations. For example, a blog `Post` and `Video` model could share a polymorphic relation to a `Tag`

model. Using a many-to-many polymorphic relation allows you to have a single list of unique tags that are shared across blog posts and videos. First, let's examine the table structure:

```
posts
  id - integer
  name - string

videos
  id - integer
  name - string

tags
  id - integer
  name - string

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

## Model Structure

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

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

## Defining The Inverse Of The Relationship

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

```
<?php
```

```

namespace App;

use Illuminate\Database\Eloquent\Model;

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

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }
}

```

## Retrieving The Relationship

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

```

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

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

```

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

```

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

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

```

## Querying Relations

Since all types of Eloquent relationships are defined via methods, you may call those methods to obtain an instance of the relationship without actually executing the relationship queries. In addition,



all types of Eloquent relationships also serve as **query builders**, allowing you to continue to chain constraints onto the relationship query before finally executing the SQL against your database.

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

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts()
    {
        return $this->hasMany('App\Post');
    }
}
```

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

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

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

You are able to use any of the **query builder** 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.

## 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:

```
$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** to pre-load relationships they know will be accessed after loading the model. Eager loading provides a significant reduction in SQL queries that must be executed to load a model's relations.

## Querying Relationship Existence

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

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

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

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

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

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

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

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

## Querying Relationship Absence

When accessing the records for a model, 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 `doesn'tHave` and `orDoesntHave` methods:

```
$posts = App\Post::doesn'tHave('comments')->get();
```

If you need even more power, you may use the `whereDoesntHave` and `orWhereDoesntHave` methods to put "where" conditions on your `doesn'tHave` queries. These methods allows you to add customized constraints to a relationship constraint, such as checking the content of a comment:

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

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

## Counting Related Models

If you want to count the number of results from a relationship without actually loading them you may use the `withCount` method, which will place a `{relation}_count` column on your resulting models. For example:

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

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

You may add the "counts" for multiple relations as well as add constraints to the queries:

```
$posts = App\Post::withCount(['votes', 'comments' => function ($query) {
    $query->where('content', 'like', 'foo%');
}])->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:

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

echo $posts[0]->comments_count;

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

## Eager Loading

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

```
<?php
```

```
namespace App;

use Illuminate\Database\Eloquent\Model;

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

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

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

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

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

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

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

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

For this operation, only two queries will be executed:

```
select * from books

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

## Eager Loading Multiple Relationships

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

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

## Nested Eager Loading

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

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

## 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:

```
$users = App\Book::with('author:id,name')->get();
```

**When using this feature, you should always include the `id` column in the list of columns you wish to retrieve.**

## Constraining Eager Loads

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

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

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

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

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

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

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

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

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

```
public function format(Book $book)
{
    $book->loadMissing('author');

    return [
        'name' => $book->name,
        'author' => $book->author->name
    ];
}
```

## Inserting & Updating Related Models

### The Save Method

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

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

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

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

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

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

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

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

## The Create Method

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

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

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

**Before using the `create` method, be sure to review the documentation on attribute mass assignment.**

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

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

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

## Belongs To Relationships

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

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

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

$user->save();
```

When removing a `belongsTo` relationship, you may use the `dissociate` method. This method will set the relationship's foreign key to `null`:

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

$user->save();
```

## Many To Many Relationships

### Attaching / Detaching

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

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

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

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

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

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

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

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

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

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

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

$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['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 do not want to detach existing IDs, 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 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]);
```

## Saving Additional Data On A Pivot Table

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]);
```

## Updating A Record On A Pivot Table

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

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

## Touching Parent Timestamps

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

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

    /**
     * Get the post that the comment belongs to.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Now, when you update a `Comment`, the owning `Post` will have its `updated_at` column updated as well, making it more convenient to know when to invalidate a cache of the `Post` model:

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

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

$comment->save();
```