Eloquent ORM

Introduction

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

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

Basic Usage

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

Defining An Eloquent Model

```
class User extends Model {}
```

You may also generate Eloquent models using the `make:model` command:

```
php artisan make:model User
```

Note that we did not tell Eloquent which table to use for our `User` model. The "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the `User` model stores records in the `users` table. You may specify a custom table by defining a `table` property on your model:

```
class User extends Model {

    protected $table = 'my_users';

}
```

> **Note:** Eloquent will also assume that each table has a primary key column named `id`. You may define a `primaryKey` property to override this convention. Likewise, you may define a `connection` property to override the name of the database connection that should be used when utilizing the model.

Once a model is defined, you are ready to start retrieving and creating records in your table. Note that you will need to place `updated_at` and `created_at` columns on your table by default. If you do not wish to have these columns automatically maintained, set the `$timestamps` property on your model to `false`.

Retrieving All Records

```
$users = User::all();
```

Retrieving A Record By Primary Key

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

var_dump($user->name);
```

> **Note:** All methods available on the query builder are also available when querying Eloquent models.

Retrieving A Model By Primary Key Or Throw An Exception

Sometimes you may wish to throw an exception if a model is not found. To do this, you may use the `firstOrFail` method:

```
$model = User::findOrFail(1);

$model = User::where('votes', '>', 100)->firstOrFail();
```

Doing this will let you catch the exception so you can log and display an error page as necessary. To catch the `ModelNotFoundException`, add some logic to your `app/Exceptions/Handler.php` file.

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

class Handler extends ExceptionHandler {

    public function render($request, Exception $e)
    {
        if ($e instanceof ModelNotFoundException)
        {
            // Custom logic for model not found...
        }

        return parent::render($request, $e);
    }

}
```

Querying Using Eloquent Models

```
$users = User::where('votes', '>', 100)->take(10)->get();

foreach ($users as $user)
{
    var_dump($user->name);
}
```

Eloquent Aggregates

Of course, you may also use the query builder aggregate functions.

```
$count = User::where('votes', '>', 100)->count();
```

If you are unable to generate the query you need via the fluent interface, feel free to use `whereRaw`:

```
$users = User::whereRaw('age > ? and votes = 100', [25])->get();
```

Chunking Results

If you need to process a lot (thousands) of Eloquent records, using the `chunk` command will allow you to do without eating all of your RAM:

```
User::chunk(200, function($users)
{
    foreach ($users as $user)
    {
        //
    }
});
```

The first argument passed to the method is the number of records you wish to receive per "chunk". The Closure passed as the second argument will be called for each chunk that is pulled from the database.

Specifying The Query Connection

You may also specify which database connection should be used when running an Eloquent query. Simply use the `on` method:

```
$user = User::on('connection-name')->find(1);
```

If you are using read / write connections, you may force the query to use the "write" connection with the following method:

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

Mass Assignment

When creating a new model, you pass an array of attributes to the model constructor. These attributes are then assigned to the model via mass-assignment. This is convenient; however, can be a **serious** security concern when blindly passing user input into a model. If user input is blindly passed into a model, the user is free to modify **any** and **all** of the model's attributes. For this reason, all Eloquent models protect against mass-assignment by default.

To get started, set the `fillable` or `guarded` properties on your model.

Defining Fillable Attributes On A Model

The `fillable` property specifies which attributes should be mass-assignable. This can be set at the class or instance level.

```
class User extends Model {
```

```
    protected $fillable = ['first_name', 'last_name', 'email'];

}
```

In this example, only the three listed attributes will be mass-assignable.

Defining Guarded Attributes On A Model

The inverse of `fillable` is `guarded`, and serves as a "black-list" instead of a "white-list":

```
class User extends Model {

    protected $guarded = ['id', 'password'];

}
```

> **Note:** When using `guarded`, you should still never pass `Input::get()` or any raw array of user controlled input into a `save` or `update` method, as any column that is not guarded may be updated.

Blocking All Attributes From Mass Assignment

In the example above, the `id` and `password` attributes may **not** be mass assigned. All other attributes will be mass assignable. You may also block **all** attributes from mass assignment using the guard property:

```
protected $guarded = ['*'];
```

Insert, Update, Delete

To create a new record in the database from a model, simply create a new model instance and call the `save` method.

Saving A New Model

```
$user = new User;

$user->name = 'John';

$user->save();
```

> **Note:** Typically, your Eloquent models will have auto-incrementing keys. However, if you wish to specify your own keys, set the `incrementing` property on your model to `false`.

You may also use the `create` method to save a new model in a single line. The inserted model instance will be returned to you from the method. However, before doing so, you will need to specify either a `fillable` or `guarded` attribute on the model, as all Eloquent models protect against mass-assignment.

After saving or creating a new model that uses auto-incrementing IDs, you may retrieve the ID by accessing the object's `id` attribute:

```
$insertedId = $user->id;
```

Setting The Guarded Attributes On The Model

```
class User extends Model {

    protected $guarded = ['id', 'account_id'];

}
```

Using The Model Create Method

```
// Create a new user in the database...
$user = User::create(['name' => 'John']);

// Retrieve the user by the attributes, or create it if it doesn't exist...
$user = User::firstOrCreate(['name' => 'John']);

// Retrieve the user by the attributes, or instantiate a new instance...
$user = User::firstOrNew(['name' => 'John']);
```

Updating A Retrieved Model

To update a model, you may retrieve it, change an attribute, and use the `save` method:

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

$user->email = 'john@foo.com';

$user->save();
```

Saving A Model And Relationships

Sometimes you may wish to save not only a model, but also all of its relationships. To do so, you may use the `push` method:

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

```
```

You may also run updates as queries against a set of models:

```
$affectedRows = User::where('votes', '>', 100)->update(['status' => 2]);
```

> **Note:** No model events are fired when updating a set of models via the Eloquent query builder.

Deleting An Existing Model

To delete a model, simply call the `delete` method on the instance:

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

$user->delete();
```

Deleting An Existing Model By Key

```
User::destroy(1);

User::destroy([1, 2, 3]);

User::destroy(1, 2, 3);
```

Of course, you may also run a delete query on a set of models:

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

Updating Only The Model's Timestamps

If you wish to simply update the timestamps on a model, you may use the `touch` method:

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

Soft Deleting

When soft deleting a model, it is not actually removed from your database. Instead, a `deleted_at` timestamp is set on the record. To enable soft deletes for a model, apply the `SoftDeletes` to the model:

```
use Illuminate\Database\Eloquent\SoftDeletes;
```

```
class User extends Model {

    use SoftDeletes;

    protected $dates = ['deleted_at'];

}
```

To add a `deleted_at` column to your table, you may use the `softDeletes` method from a migration:

```
$table->softDeletes();
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current timestamp. When querying a model that uses soft deletes, the "deleted" models will not be included in query results.

Forcing Soft Deleted Models Into Results

To force soft deleted models to appear in a result set, use the `withTrashed` method on the query:

```
$users = User::withTrashed()->where('account_id', 1)->get();
```

The `withTrashed` method may be used on a defined relationship:

```
$user->posts()->withTrashed()->get();
```

If you wish to **only** receive soft deleted models in your results, you may use the `onlyTrashed` method:

```
$users = User::onlyTrashed()->where('account_id', 1)->get();
```

To restore a soft deleted model into an active state, use the `restore` method:

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

You may also use the `restore` method on a query:

```
User::withTrashed()->where('account_id', 1)->restore();
```

Like with `withTrashed`, the `restore` method may also be used on relationships:

```
$user->posts()->restore();
```

If you wish to truly remove a model from the database, you may use the `forceDelete` method:

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

The `forceDelete` method also works on relationships:

```
$user->posts()->forceDelete();
```

To determine if a given model instance has been soft deleted, you may use the `trashed` method:

```
if ($user->trashed())
{
    //
}
```

Timestamps

By default, Eloquent will maintain the `created_at` and `updated_at` columns on your database table automatically. Simply add these `timestamp` columns to your table and Eloquent will take care of the rest. If you do not wish for Eloquent to maintain these columns, add the following property to your model:

Disabling Auto Timestamps

```
class User extends Model {

    protected $table = 'users';

    public $timestamps = false;

}
```

Providing A Custom Timestamp Format

If you wish to customize the format of your timestamps, you may override the `getDateFormat` method in your model:

```
class User extends Model {

    protected function getDateFormat()
    {
        return 'U';
    }
```

```
}
```

Query Scopes

Defining A Query Scope

Scopes allow you to easily re-use query logic in your models. To define a scope, simply prefix a model method with `scope`:

```
class User extends Model {

    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }

}
```

Utilizing A Query Scope

```
$users = User::popular()->women()->orderBy('created_at')->get();
```

Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. Just add your parameters to your scope function:

```
class User extends Model {

    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }

}
```

Then pass the parameter into the scope call:

```
$users = User::ofType('member')->get();
```

Global Scopes

Sometimes you may wish to define a scope that applies to all queries performed on a model. In essence, this is how Eloquent's own "soft delete" feature works. Global scopes are defined using a combination of PHP traits and an implementation of `Illuminate\Database\Eloquent\ScopeInterface`.

First, let's define a trait. For this example, we'll use the `SoftDeletes` that ships with Laravel:

```
trait SoftDeletes {

    /**
     * Boot the soft deleting trait for a model.
     *
     * @return void
     */
    public static function bootSoftDeletes()
    {
        static::addGlobalScope(new SoftDeletingScope);
    }

}
```

If an Eloquent model uses a trait that has a method matching the `bootNameOfTrait` naming convention, that trait method will be called when the Eloquent model is booted, giving you an opportunity to register a global scope, or do anything else you want. A scope must implement `ScopeInterface`, which specifies two methods: `apply` and `remove`.

The `apply` method receives an `Illuminate\Database\Eloquent\Builder` query builder object and the `Model` it's applied to, and is responsible for adding any additional `where` clauses that the scope wishes to add. The `remove` method also receives a `Builder` object and `Model` and is responsible for reversing the action taken by `apply`. In other words, `remove` should remove the `where` clause (or any other clause) that was added. So, for our `SoftDeletingScope`, the methods look something like this:

```
/**
 * Apply the scope to a given Eloquent query builder.
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $builder
 * @param  \Illuminate\Database\Eloquent\Model  $model
 * @return void
 */
public function apply(Builder $builder, Model $model)
{
    $builder->whereNull($model->getQualifiedDeletedAtColumn());

    $this->extend($builder);
}

/**
 * Remove the scope from the given Eloquent query builder.
```

```
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $builder
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @return void
     */
    public function remove(Builder $builder, Model $model)
    {
        $column = $model->getQualifiedDeletedAtColumn();

        $query = $builder->getQuery();

        foreach ((array) $query->wheres as $key => $where)
        {
            // If the where clause is a soft delete date constraint, we will remove it
from
            // the query and reset the keys on the wheres. This allows this developer
to
            // include deleted model in a relationship result set that is lazy loaded.
            if ($this->isSoftDeleteConstraint($where, $column))
            {
                unset($query->wheres[$key]);

                $query->wheres = array_values($query->wheres);
            }
        }
    }
```

Relationships

Of course, your database tables are probably 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. Laravel supports many types of relationships:

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

One To One

Defining A One To One Relation

A one-to-one relationship is a very basic relation. For example, a `User` model might have one `Phone`. We can define this relation in Eloquent:

```
class User extends Model {
```

```
    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 it using Eloquent's dynamic properties:

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

The SQL performed by this statement will be as follows:

```
select * from users where id = 1

select * from phones where user_id = 1
```

Take note that Eloquent assumes the foreign key of the relationship based on the model name. In this case, `Phone` model is assumed to use a `user_id` foreign key. If you wish to override this convention, you may pass a second argument to the `hasOne` method. Furthermore, you may pass a third argument to the method to specify which local column that should be used for the association:

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

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

Defining The Inverse Of A Relation

To define the inverse of the relationship on the `Phone` model, we use the `belongsTo` method:

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User');
    }

}
```

In the example above, Eloquent will look for a `user_id` column on the `phones` table. If you would like to define a different foreign key column, you may pass it as the second argument to the `belongsTo` method:

```
class Phone extends Model {
```

```
    public function user()
    {
        return $this->belongsTo('App\User', 'local_key');
    }

}
```

Additionally, you pass a third parameter which specifies the name of the associated column on the parent table:

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User', 'local_key', 'parent_key');
    }

}
```

One To Many

An example of a one-to-many relation is a blog post that "has many" comments. We can model this relation like so:

```
class Post extends Model {

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

}
```

Now we can access the post's comments through the dynamic property:

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

If you need to add further constraints to which comments are retrieved, you may call the `comments` method and continue chaining conditions:

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

Again, you may override the conventional foreign key by passing a second argument to the `hasMany` method. And, like the `hasOne` relation, the local column may also be specified:

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

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

Defining The Inverse Of A Relation

To define the inverse of the relationship on the `Comment` model, we use the `belongsTo` method:

```
class Comment extends Model {

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

Many To Many

Many-to-many relations are a more complicated relationship type. 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". Three database tables are needed for this relationship: `users`, `roles`, and `role_user`. The `role_user` table is derived from the alphabetical order of the related model names, and should have `user_id` and `role_id` columns.

We can define a many-to-many relation using the `belongsToMany` method:

```
class User extends Model {

    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }

}
```

Now, we can retrieve the roles through the `User` model:

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

If you would like to use an unconventional table name for your pivot table, you may pass it as the second argument to the `belongsToMany` method:

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

```
```

You may also override the conventional associated keys:

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

Of course, you may also define the inverse of the relationship on the `Role` model:

```
class Role extends Model {

    public function users()
    {
        return $this->belongsToMany('App\User');
    }

}
```

Has Many Through

The "has many through" relation provides a convenient short-cut for accessing distant relations via an intermediate relation. For example, a `Country` model might have many `Post` through a `User` model. The tables for this relationship would look like this:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

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

Even though the `posts` table does not contain a `country_id` column, the `hasManyThrough` relation will allow us to access a country's posts via `$country->posts`. Let's define the relationship:

```
class Country extends Model {

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

}
```

If you would like to manually specify the keys of the relationship, you may pass them as the third and fourth arguments to the method:

```
class Country extends Model {

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

}
```

Polymorphic Relations

Polymorphic relations allow a model to belong to more than one other model, on a single association. For example, you might have a photo model that belongs to either a staff model or an order model. We would define this relation like so:

```
class Photo extends Model {

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

}

class Staff extends Model {

    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }

}

class Order extends Model {

    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }

}
```

Retrieving A Polymorphic Relation

Now, we can retrieve the photos for either a staff member or an order:

```
$staff = Staff::find(1);

foreach ($staff->photos as $photo)
{
    //
}
```

Retrieving The Owner Of A Polymorphic Relation

However, the true "polymorphic" magic is when you access the staff or order from the `Photo` model:

```
$photo = Photo::find(1);

$imageable = $photo->imageable;
```

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

Polymorphic Relation Table Structure

To help understand how this works, let's explore the database structure for a polymorphic relation:

```
staff
    id - integer
    name - string

orders
    id - integer
    price - integer

photos
    id - integer
    path - string
    imageable_id - integer
    imageable_type - string
```

The key fields to notice here are the `imageable_id` and `imageable_type` on the `photos` table. The ID will contain the ID value of, in this example, the owning staff or order, while the type will contain the class name of the owning model. This is what allows the ORM to determine which type of owning model to return when accessing the `imageable` relation.

Many To Many Polymorphic Relations

Polymorphic Many To Many Relation Table Structure

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

model. 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
```

Next, we're ready to setup the relationships on the model. The `Post` and `Video` model will both have a `morphToMany` relationship via a `tags` method:

```
class Post extends Model {

    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }

}
```

The `Tag` model may define a method for each of its relationships:

```
class Tag extends Model {

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

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

}
```

Querying Relations

Querying Relations When Selecting

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

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

You may also specify an operator and a count:

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

Nested `has` statements may also be constructed using "dot" notation:

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

If you need even more power, you may use the `whereHas` and `orWhereHas` methods to put "where" conditions on your `has` queries:

```
$posts = Post::whereHas('comments', function($q)
{
    $q->where('content', 'like', 'foo%');

})->get();
```

Dynamic Properties

Eloquent allows you to access your relations via dynamic properties. Eloquent will automatically load the relationship for you, and is even smart enough to know whether to call the `get` (for one-to-many relationships) or `first` (for one-to-one relationships) method. It will then be accessible via a dynamic property by the same name as the relation. For example, with the following model `$phone`:

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User');
    }

}

$phone = Phone::find(1);
```

Instead of echoing the user's email like this:

```
echo $phone->user()->first()->email;
```

It may be shortened to simply:

```
echo $phone->user->email;
```

> **Note:** Relationships that return many results will return an instance of the `Illuminate\Database\Eloquent\Collection` class.

Eager Loading

Eager loading exists to alleviate the N + 1 query problem. For example, consider a `Book` model that is related to `Author`. The relationship is defined like so:

```
class Book extends Model {

    public function author()
    {
        return $this->belongsTo('App\Author');
    }

}
```

Now, consider the following code:

```
foreach (Book::all() 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.

Thankfully, we can use eager loading to drastically reduce the number of queries. The relationships that should be eager loaded may be specified via the `with` method:

```
foreach (Book::with('author')->get() as $book)
{
    echo $book->author->name;
```

```
}
```

In the loop above, only two queries will be executed:

```
select * from books

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

Wise use of eager loading can drastically increase the performance of your application.

Of course, you may eager load multiple relationships at one time:

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

You may even eager load nested relationships:

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

In the example above, the `author` relationship will be eager loaded, and the author's `contacts` relation will also be loaded.

Eager Load Constraints

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

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

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

In this example, we're eager loading the user's posts, but only if the post's title column contains the word "first".

Of course, eager loading Closures aren't limited to "constraints". You may also apply orders:

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

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

Lazy Eager Loading

It is also possible to eagerly load related models directly from an already existing model collection. This may be useful when dynamically deciding whether to load related models or not, or in combination with caching.

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

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

You may also pass a Closure to set constraints on the query:

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

Inserting Related Models

Attaching A Related Model

You will often need to insert new related models. For example, you may wish to insert a new comment for a post. Instead of manually setting the `post_id` foreign key on the model, you may insert the new comment from its parent `Post` model directly:

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

$post = Post::find(1);

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

In this example, the `post_id` field will automatically be set on the inserted comment.

If you need to save multiple related models:

```
$comments = [
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another comment.']),
    new Comment(['message' => 'The latest comment.'])
];

$post = Post::find(1);

$post->comments()->saveMany($comments);
```

Associating Models (Belongs To)

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

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

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

$user->save();
```

Inserting Related Models (Many To Many)

You may also insert related models when working with many-to-many relations. Let's continue using our `User` and `Role` models as examples. We can easily attach new roles to a user using the `attach` method:

Attaching Many To Many Models

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

$user->roles()->attach(1);
```

You may also pass an array of attributes that should be stored on the pivot table for the relation:

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

Of course, the opposite of `attach` is `detach`:

```
$user->roles()->detach(1);
```

Both `attach` and `detach` also take arrays of IDs as input:

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

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

$user->roles()->attach([1 => ['attribute1' => 'value1'], 2, 3]);
```

Using Sync To Attach Many To Many Models

You may also use the `sync` method to attach related models. The `sync` method accepts an array of IDs to place on the pivot table. After this operation is complete, only the IDs in the array will be on the intermediate table for the model:

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

Adding Pivot Data When Syncing

You may also associate other pivot table values with the given IDs:

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

Sometimes you may wish to create a new related model and attach it in a single command. For this operation, you may use the `save` method:

```
$role = new Role(['name' => 'Editor']);

User::find(1)->roles()->save($role);
```

In this example, the new `Role` model will be saved and attached to the user model. You may also pass an array of attributes to place on the joining table for this operation:

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

Touching Parent Timestamps

When a model `belongsTo` another model, such as a `Comment` which belongs to a `Post`, it is often 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:

```
class Comment extends Model {

    protected $touches = ['post'];

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

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

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

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

$comment->save();
```

Working With Pivot Tables

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 `pivot` table on the models:

```
$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, and may be used as any other Eloquent model.

By default, only the 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('foo', 'bar');
```

Now the `foo` and `bar` attributes will be accessible on our `pivot` object for the `Role` model.

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();
```

Deleting Records On A Pivot Table

To delete all records on the pivot table for a model, you may use the `detach` method:

```
User::find(1)->roles()->detach();
```

Note that this operation does not delete records from the `roles` table, but only from the pivot table.

Updating A Record On A Pivot Table

Sometimes you may need to update your pivot table, but not detach it. If you wish to update your pivot table in place you may use `updateExistingPivot` method like so:

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

Defining A Custom Pivot Model

Laravel also allows you to define a custom Pivot model. To define a custom model, first create your own "Base" model class that extends `Eloquent`. In your other Eloquent models, extend this custom base model instead of the default `Eloquent` base. In your base model, add the following function that returns an instance of your custom Pivot model:

```
public function newPivot(Model $parent, array $attributes, $table, $exists)
{
    return new YourCustomPivot($parent, $attributes, $table, $exists);
}
```

Collections

All multi-result sets returned by Eloquent, either via the `get` method or a `relationship`, will return a collection object. This object implements the `IteratorAggregate` PHP interface so it can be iterated over like an array. However, this object also has a variety of other helpful methods for working with result sets.

Checking If A Collection Contains A Key

For example, we may determine if a result set contains a given primary key using the `contains` method:

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

if ($roles->contains(2))
{
    //
}
```

Collections may also be converted to an array or JSON:

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

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

If a collection is cast to a string, it will be returned as JSON:

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

Iterating Collections

Eloquent collections also contain a few helpful methods for looping and filtering the items they contain:

```
$roles = $user->roles->each(function($role)
{
    //
});
```

Filtering Collections

When filtering collections, the callback provided will be used as callback for array_filter.

```
$users = $users->filter(function($user)
{
    return $user->isAdmin();
});
```

> **Note:** When filtering a collection and converting it to JSON, try calling the `values` function first to reset the array's keys.

Applying A Callback To Each Collection Object

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

$roles->each(function($role)
{
    //
});
```

Sorting A Collection By A Value

```
$roles = $roles->sortBy(function($role)
{
    return $role->created_at;
});

$roles = $roles->sortByDesc(function($role)
{
    return $role->created_at;
});
```

Sorting A Collection By A Value

```
$roles = $roles->sortBy('created_at');

$roles = $roles->sortByDesc('created_at');
```

Returning A Custom Collection Type

Sometimes, you may wish to return a custom Collection object with your own added methods. You may specify this on your Eloquent model by overriding the `newCollection` method:

```
class User extends Model {

    public function newCollection(array $models = [])
    {
        return new CustomCollection($models);
    }

}
```

Accessors & Mutators

Defining An Accessor

Eloquent provides a convenient way to transform your model attributes when getting or setting them. Simply define a `getFooAttribute` method on your model to declare an accessor. Keep in mind that the methods should follow camel-casing, even though your database columns are snake-case:

```
class User extends Model {

    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }

}
```

In the example above, the `first_name` column has an accessor. Note that the value of the attribute is passed to the accessor.

Defining A Mutator

Mutators are declared in a similar fashion:

```
class User extends Model {
```

```
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }

}
```

Date Mutators

By default, Eloquent will convert the `created_at` and `updated_at` columns to instances of Carbon, which provides an assortment of helpful methods, and extends the native PHP `DateTime` class.

You may customize which fields are automatically mutated, and even completely disable this mutation, by overriding the `getDates` method of the model:

```
public function getDates()
{
    return ['created_at'];
}
```

When a column is considered a date, you may set its value to a UNIX timestamp, date string (`Y-m-d`), date-time string, and of course a `DateTime` / `Carbon` instance.

To totally disable date mutations, simply return an empty array from the `getDates` method:

```
public function getDates()
{
    return [];
}
```

Attribute Casting

If you have some attributes that you want to always convert to another data-type, you may add the attribute to the `casts` property of your model. Otherwise, you will have to define a mutator for each of the attributes, which can be time consuming. Here is an example of using the `casts` property:

```
/**
 * The attributes that should be casted to native types.
 *
 * @var array
 */
protected $casts = [
    'is_admin' => 'boolean',
];
```

Now the `is_admin` attribute will always be cast to a boolean when you access it, even if the underlying value is stored in the database as an integer. Other supported cast types are: `integer`, `real`, `float`, `double`, `string`, `boolean`, `object` and `array`.

The `array` cast is particularly useful for working with columns that are stored as serialized JSON. For example, if your database has a TEXT type field that contains serialized JSON, adding the `array` cast to that attribute will automatically deserialize the attribute to a PHP array when you access it on your Eloquent model:

```
/**
 * The attributes that should be casted to native types.
 *
 * @var array
 */
protected $casts = [
    'options' => 'array',
];
```

Now, when you utilize the Eloquent model:

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

// $options is an array...
$options = $user->options;

// options is automatically serialized back to JSON...
$user->options = ['foo' => 'bar'];
```

Model Events

Eloquent models fire several events, allowing you to hook into various points in the model's lifecycle using the following methods: `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`.

Whenever a new item is saved for the first time, the `creating` and `created` events will fire. If an item is not new and the `save` method is called, the `updating` / `updated` events will fire. In both cases, the `saving` / `saved` events will fire.

Cancelling Save Operations Via Events

If `false` is returned from the `creating`, `updating`, `saving`, or `deleting` events, the action will be cancelled:

```
User::creating(function($user)
{
    if ( ! $user->isValid()) return false;
```

```
});
```

Where To Register Event Listeners

Your `EventServiceProvider` serves as a convenient place to register your model event bindings. For example:

```
/**
 * Register any other events for your application.
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    User::creating(function($user)
    {
        //
    });
}
```

Model Observers

To consolidate the handling of model events, you may register a model observer. An observer class may have methods that correspond to the various model events. For example, `creating`, `updating`, `saving` methods may be on an observer, in addition to any other model event name.

So, for example, a model observer might look like this:

```
class UserObserver {

    public function saving($model)
    {
        //
    }

    public function saved($model)
    {
        //
    }

}
```

You may register an observer instance using the `observe` method:

```
User::observe(new UserObserver);
```

Model URL Generation

When you pass a model to the `route` or `action` methods, it's primary key is inserted into the generated URI. For example:

```
Route::get('user/{user}', 'UserController@show');

action('UserController@show', [$user]);
```

In this example the `$user->id` property will be inserted into the `{user}` place-holder of the generated URL. However, if you would like to use another property instead of the ID, you may override the `getRouteKey` method on your model:

```
public function getRouteKey()
{
    return $this->slug;
}
```

Converting To Arrays / JSON

Converting A Model To An Array

When building JSON APIs, you may often need to convert your models and relationships to arrays or JSON. So, Eloquent includes methods for doing so. To convert a model and its loaded relationship to an array, you may use the `toArray` method:

```
$user = User::with('roles')->first();

return $user->toArray();
```

Note that entire collections of models may also be converted to arrays:

```
return User::all()->toArray();
```

Converting A Model To JSON

To convert a model to JSON, you may use the `toJson` method:

```
return User::find(1)->toJson();
```

Returning A Model From A Route

Note that when a model or collection is cast to a string, it will be converted to JSON, meaning you can return Eloquent objects directly from your application's routes!

```
Route::get('users', function()
{
    return User::all();
});
```

Hiding Attributes From Array Or JSON Conversion

Sometimes you may wish to limit the attributes that are included in your model's array or JSON form, such as passwords. To do so, add a `hidden` property definition to your model:

```
class User extends Model {

    protected $hidden = ['password'];

}
```

> **Note:** When hiding relationships, use the relationship's **method** name, not the dynamic accessor name.

Alternatively, you may use the `visible` property to define a white-list:

```
protected $visible = ['first_name', 'last_name'];
```

Occasionally, you may need to add array attributes that do not have a corresponding column in your database. To do so, simply define an accessor for the value:

```
public function getIsAdminAttribute()
{
    return $this->attributes['admin'] == 'yes';
}
```

Once you have created the accessor, just add the value to the `appends` property on the model:

```
protected $appends = ['is_admin'];
```

Once the attribute has been added to the `appends` list, it will be included in both the model's array and JSON forms. Attributes in the `appends` array respect the `visible` and `hidden` configuration on the model.