Eloquent: Getting Started

Introduction

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.

> [!NOTE] Before getting started, be sure to configure a database connection in your application's `config/database.php` configuration file. For more information on configuring your database, check out the database configuration documentation.

Generating Model Classes

To get started, let's create an Eloquent model. Models typically live in the `app\Models` directory and extend the `Illuminate\Database\Eloquent\Model` class. You may use the `make:model` Artisan command to generate a new model:

```
php artisan make:model Flight
```

If you would like to generate a database migration when you generate the model, you may use the `--migration` or `-m` option:

```
php artisan make:model Flight --migration
```

You may generate various other types of classes when generating a model, such as factories, seeders, policies, controllers, and form requests. In addition, these options may be combined to create multiple classes at once:

```
# Generate a model and a FlightFactory class...
php artisan make:model Flight --factory
php artisan make:model Flight -f

# Generate a model and a FlightSeeder class...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# Generate a model and a FlightController class...
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Generate a model, FlightController resource class, and form request classes...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR
```

```
# Generate a model and a FlightPolicy class...
php artisan make:model Flight --policy

# Generate a model and a migration, factory, seeder, and controller...
php artisan make:model Flight -mfsc

# Shortcut to generate a model, migration, factory, seeder, policy, controller,
and form requests...
php artisan make:model Flight --all
php artisan make:model Flight -a

# Generate a pivot model...
php artisan make:model Member --pivot
php artisan make:model Member -p
```

Inspecting Models

Sometimes it can be difficult to determine all of a model's available attributes and relationships just by skimming its code. Instead, try the `model:show` Artisan command, which provides a convenient overview of all the model's attributes and relations:

```
php artisan model:show Flight
```

Eloquent Model Conventions

Models generated by the `make:model` command will be placed in the `app/Models` directory. Let's examine a basic model class and discuss some of Eloquent's key conventions:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    // ...
}
```

Table Names

After glancing at the example above, you may have noticed that we did not tell Eloquent which database table corresponds to our `Flight` model. By convention, 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 `Flight` model stores records in the `flights` table, while an `AirTrafficController` model would store records in an `air_traffic_controllers` table.

If your model's corresponding database table does not fit this convention, you may manually specify the model's table name by defining a `table` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Primary Keys

Eloquent will also assume that each model's corresponding database table has a primary key column named `id`. If necessary, you may define a protected `$primaryKey` property on your model to specify a different column that serves as your model's primary key:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that Eloquent will automatically cast the primary key to an integer. If you wish to use a non-incrementing or a non-numeric primary key you must define a public `$incrementing` property on your model that is set to `false`:

```php
<?php

class Flight extends Model
{
    /**
     * Indicates if the model's ID is auto-incrementing.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

If your model's primary key is not an integer, you should define a protected `$keyType` property on your model. This property should have a value of `string`:

```php
<?php

class Flight extends Model
{
    /**
     * The data type of the primary key ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

"Composite" Primary Keys

Eloquent requires each model to have at least one uniquely identifying "ID" that can serve as its primary key. "Composite" primary keys are not supported by Eloquent models. However, you are free to add additional multi-column, unique indexes to your database tables in addition to the table's uniquely identifying primary key.

UUID and ULID Keys

Instead of using auto-incrementing integers as your Eloquent model's primary keys, you may choose to use UUIDs instead. UUIDs are universally unique alpha-numeric identifiers that are 36 characters long.

If you would like a model to use a UUID key instead of an auto-incrementing integer key, you may use the `Illuminate\Database\Eloquent\Concerns\HasUuids` trait on the model. Of course, you should ensure that the model has a UUID equivalent primary key column:

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

class Article extends Model
{
    use HasUuids;

    // ...
}

$article = Article::create(['title' => 'Traveling to Europe']);

$article->id; // "8f8e8478-9035-4d23-b9a7-62f4d2612ce5"
```

By default, The `HasUuids` trait will generate "ordered" UUIDs for your models. These UUIDs are more efficient for indexed database storage because they can be sorted lexicographically.

You can override the UUID generation process for a given model by defining a `newUniqueId` method on the model. In addition, you may specify which columns should receive UUIDs by defining a `uniqueIds` method on the model:

```
use Ramsey\Uuid\Uuid;
/**
 * Generate a new UUID for the model.
 */
public function newUniqueId(): string
{
    return (string) Uuid::uuid4();
}

/**
 * Get the columns that should receive a unique identifier.
 *
 * @return array<int, string>
 */
public function uniqueIds(): array
{
    return ['id', 'discount_code'];
}
```

If you wish, you may choose to utilize "ULIDs" instead of UUIDs. ULIDs are similar to UUIDs; however, they are only 26 characters in length. Like ordered UUIDs, ULIDs are lexicographically sortable for efficient database indexing. To utilize ULIDs, you should use the `Illuminate\Database\Eloquent\Concerns\HasUlids` trait on your model. You should also ensure that the model has a ULID equivalent primary key column:

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

class Article extends Model
{
    use HasUlids;

    // ...
}

$article = Article::create(['title' => 'Traveling to Asia']);

$article->id; // "01gd4d3tgrrfqeda94gdbtdk5c"
```

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your model's corresponding database table. Eloquent will automatically set these column's values when models are created or updated. If you do not want these columns to be automatically managed by Eloquent, you should define a `$timestamps` property on your model with a value of `false`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

If you need to customize the format of your model's timestamps, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database as well as their format when the model is serialized to an array or JSON:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

If you need to customize the names of the columns used to store the timestamps, you may define `CREATED_AT` and `UPDATED_AT` constants on your model:

```php
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

If you would like to perform model operations without the model having its `updated_at` timestamp modified, you may operate on the model within a closure given to the `withoutTimestamps` method:

```php
Model::withoutTimestamps(fn () => $post->increment('reads'));
```

Database Connections

By default, all Eloquent models will use the default database connection that is configured for your application. If you would like to specify a different connection that should be used when interacting with a particular model, you should define a `$connection` property on the model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The database connection that should be used by the model.
     *
     * @var string
     */
    protected $connection = 'mysql';
}
```

Default Attribute Values

By default, a newly instantiated model instance will not contain any attribute values. If you would like to define the default values for some of your model's attributes, you may define an `$attributes` property on your model. Attribute values placed in the `$attributes` array should be in their raw, "storable" format as if they were just read from the database:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The model's default values for attributes.
     *
     * @var array
     */
    protected $attributes = [
        'options' => '[]',
        'delayed' => false,
    ];
}
```

Configuring Eloquent Strictness

Laravel offers several methods that allow you to configure Eloquent's behavior and "strictness" in a variety of situations.

First, 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. Typically, this method should be invoked in the `boot` method of your application's `AppServiceProvider`:

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

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

Also, you may instruct Laravel to throw an exception when attempting to fill an unfillable attribute by invoking the `preventSilentlyDiscardingAttributes` method. This can help prevent unexpected errors during local development when attempting to set an attribute that has not been added to the model's `fillable` array:

```
Model::preventSilentlyDiscardingAttributes(! $this->app->isProduction());
```

Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. You can think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. The model's `all` method will retrieve all of the records from the model's associated database table:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

Building Queries

The Eloquent `all` method will return all of the results in the model's table. However, since each Eloquent model serves as a query builder, you may add additional constraints to queries and then invoke the `get` method to retrieve the results:

```
$flights = Flight::where('active', 1)
    ->orderBy('name')
    ->take(10)
    ->get();
```

> [!NOTE] Since Eloquent models are query builders, you should review all of the methods provided by Laravel's query builder. You may use any of these methods when writing your Eloquent queries.

Refreshing Models

If you already have an instance of an Eloquent model that was retrieved from the database, you can "refresh" the model using the `fresh` and `refresh` methods. The `fresh` method will re-retrieve the model from the database. The existing model instance will not be affected:

```
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

The `refresh` method will re-hydrate the existing model using fresh data from the database. In addition, all of its loaded relationships will be refreshed as well:

```
$flight = Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

Collections

As we have seen, Eloquent methods like `all` and `get` retrieve multiple records from the database. However, these methods don't return a plain PHP array. Instead, an instance of `Illuminate\Database\Eloquent\Collection` is returned.

The Eloquent `Collection` class extends Laravel's base `Illuminate\Support\Collection` class, which provides a variety of helpful methods for interacting with data collections. For example, the `reject` method may be used to remove models from a collection based on the results of an invoked closure:

```
$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function (Flight $flight) {
    return $flight->cancelled;
});
```

In addition to the methods provided by Laravel's base collection class, the Eloquent collection class provides a few extra methods that are specifically intended for interacting with collections of Eloquent models.

Since all of Laravel's collections implement PHP's iterable interfaces, you may loop over collections as if they were an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Chunking Results

Your application may run out of memory if you attempt to load tens of thousands of Eloquent records via the `all` or `get` methods. Instead of using these methods, the `chunk` method may be used to process large numbers of models more efficiently.

The `chunk` method will retrieve a subset of Eloquent models, passing them to a closure for processing. Since only the current chunk of Eloquent models is retrieved at a time, the `chunk` method will provide significantly reduced memory usage when working with a large number of models:

```
use App\Models\Flight;
use Illuminate\Database\Eloquent\Collection;

Flight::chunk(200, function (Collection $flights) {
    foreach ($flights as $flight) {
        // ...
    }
});
```

The first argument passed to the `chunk` method is the number of records you wish to receive per "chunk". The closure passed as the second argument will be invoked for each chunk that is retrieved from the database. A database query will be executed to retrieve each chunk of records passed to the closure.

If you are filtering the results of the `chunk` method based on a column that you will also be updating while iterating over the results, you should use the `chunkById` method. Using the `chunk` method in these scenarios could lead to unexpected and inconsistent results. Internally, the `chunkById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
    ->chunkById(200, function (Collection $flights) {
        $flights->each->update(['departed' => false]);
    }, column: 'id');
```

Since the `chunkById` and `lazyById` methods add their own "where" conditions to the query being executed, you should typically logically group your own conditions within a closure:

```
Flight::where(function ($query) {
    $query->where('delayed', true)->orWhere('cancelled', true);
})->chunkById(200, function (Collection $flights) {
    $flights->each->update([
        'departed' => false,
        'cancelled' => true
    ]);
}, column: 'id');
```

Chunking Using Lazy Collections

The `lazy` method works similarly to the `chunk` method in the sense that, behind the scenes, it executes the query in chunks. However, instead of passing each chunk directly into a callback as is, the `lazy` method returns a flattened `LazyCollection` of Eloquent models, which lets you interact with the results as a single stream:

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
    // ...
}
```

If you are filtering the results of the `lazy` method based on a column that you will also be updating while iterating over the results, you should use the `lazyById` method. Internally, the `lazyById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
    ->lazyById(200, column: 'id')
    ->each->update(['departed' => false]);
```

You may filter the results based on the descending order of the `id` using the `lazyByIdDesc` method.

Cursors

Similar to the `lazy` method, the `cursor` method may be used to significantly reduce your application's memory consumption when iterating through tens of thousands of Eloquent model records.

The `cursor` method will only execute a single database query; however, the individual Eloquent models will not be hydrated until they are actually iterated over. Therefore, only one Eloquent model is kept in memory at any given time while iterating over the cursor.

> [!WARNING] Since the `cursor` method only ever holds a single Eloquent model in memory at a time, it cannot eager load relationships. If you need to eager load relationships, consider using the `lazy` method instead.

Internally, the `cursor` method uses PHP generators to implement this functionality:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
    // ...
}
```

The `cursor` returns an `Illuminate\Support\LazyCollection` instance. Lazy collections allow you to use many of the collection methods available on typical Laravel collections while only loading a single model into memory at a time:

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Although the `cursor` method uses far less memory than a regular query (by only holding a single Eloquent model in memory at a time), it will still eventually run out of memory. This is due to PHP's PDO driver internally caching all raw query results in its buffer. If you're dealing with a very large number of Eloquent records, consider using the `lazy` method instead.

Advanced Subqueries

Subquery Selects

Eloquent also offers advanced subquery support, which allows you to pull information from related tables in a single query. For example, let's imagine that we have a table of flight `destinations` and a table of `flights` to destinations. The `flights` table contains an `arrived_at` column which indicates when the flight arrived at the destination.

Using the subquery functionality available to the query builder's `select` and `addSelect` methods, we can select all of the `destinations` and the name of the flight that most recently arrived at that destination using a single query:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
])->get();
```

Subquery Ordering

In addition, the query builder's `orderBy` function supports subqueries. Continuing to use our flight example, we may use this functionality to sort all destinations based on when the last flight arrived at that destination. Again, this may be done while executing a single database query:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

Retrieving Single Models / Aggregates

In addition to retrieving all of the records matching a given query, you may also retrieve single records using the `find`, `first`, or `firstWhere` methods. Instead of returning a collection of models, these methods return a single model instance:

```
use App\Models\Flight;

// Retrieve a model by its primary key...
$flight = Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();

// Alternative to retrieving the first model matching the query constraints...
$flight = Flight::firstWhere('active', 1);
```

Sometimes you may wish to perform some other action if no results are found. The `findOr` and `firstOr` methods will return a single model instance or, if no results are found, execute the given closure. The value returned by the closure will be considered the result of the method:

```
$flight = Flight::findOr(1, function () {
    // ...
});

$flight = Flight::where('legs', '>', 3)->firstOr(function () {
    // ...
});
```

Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query; however, if no result is found, an `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

If the `ModelNotFoundException` is not caught, a 404 HTTP response is automatically sent back to the client:

```
use App\Models\Flight;

Route::get('/api/flights/{id}', function (string $id) {
    return Flight::findOrFail($id);
});
```

Retrieving or Creating Models

The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model cannot be found in the database, a record will be inserted with the attributes resulting from merging the first array argument with the optional second array argument:

The `firstOrNew` method, like `firstOrCreate`, will attempt to locate a record in the database matching the given attributes. However, if a model is not found, a new model instance will be returned. Note that the model returned by `firstOrNew` has not yet been persisted to the database. You will need to manually call the `save` method to persist it:

```
use App\Models\Flight;

// Retrieve flight by name or create it if it doesn't exist...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or create it with the name, delayed, and arrival_time
attributes...
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or instantiate with the name, delayed, and arrival_time
attributes...
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

Retrieving Aggregates

When interacting with Eloquent models, you may also use the `count`, `sum`, `max`, and other aggregate methods provided by the Laravel query builder. As you might expect, these methods return a scalar value instead of an Eloquent model instance:

```
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

Inserting and Updating Models

Inserts

Of course, when using Eloquent, we don't only need to retrieve models from the database. We also need to insert new records. Thankfully, Eloquent makes it simple. To insert a new record into the database, you should instantiate a new model instance and set attributes on the model. Then, call the `save` method on the model instance:

```
<?php

namespace App\Http\Controllers;

use App\Models\Flight;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Store a new flight in the database.
     */
    public function store(Request $request): RedirectResponse
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();

        return redirect('/flights');
    }
}
```

In this example, we assign the `name` field from the incoming HTTP request to the `name` attribute of the `App\Models\Flight` model instance. When we call the `save` method, a record will be inserted into the

database. The model's `created_at` and `updated_at` timestamps will automatically be set when the `save` method is called, so there is no need to set them manually.

Alternatively, you may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the `create` method:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default. To learn more about mass assignment, please consult the mass assignment documentation.

Updates

The `save` method may also be used to update models that already exist in the database. To update a model, you should retrieve it and set any attributes you wish to update. Then, you should call the model's `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();
```

Occasionally, you may need to update an existing model or create a new model if no matching model exists. Like the `firstOrCreate` method, the `updateOrCreate` method persists the model, so there's no need to manually call the `save` method.

In the example below, if a flight exists with a `departure` location of `Oakland` and a `destination` location of `San Diego`, its `price` and `discounted` columns will be updated. If no such flight exists, a new flight will be created which has the attributes resulting from merging the first argument array with the second argument array:

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

Mass Updates

Updates can also be performed against models that match a given query. In this example, all flights that are `active` and have a `destination` of `San Diego` will be marked as delayed:

```
Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);
```

The `update` method expects an array of column and value pairs representing the columns that should be updated. The `update` method returns the number of affected rows.

> [!WARNING] When issuing a mass update via Eloquent, the `saving`, `saved`, `updating`, and `updated` model events will not be fired for the updated models. This is because the models are never actually retrieved when issuing a mass update.

Examining Attribute Changes

Eloquent provides the `isDirty`, `isClean`, and `wasChanged` methods to examine the internal state of your model and determine how its attributes have changed from when the model was originally retrieved.

The `isDirty` method determines if any of the model's attributes have been changed since the model was retrieved. You may pass a specific attribute name or an array of attributes to the `isDirty` method to determine if any of the attributes are "dirty". The `isClean` method will determine if an attribute has remained unchanged since the model was retrieved. This method also accepts an optional attribute argument:

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false
$user->isDirty(['first_name', 'title']); // true

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true
$user->isClean(['first_name', 'title']); // false
$user->save();
$user->isDirty(); // false
$user->isClean(); // true
```

The `wasChanged` method determines if any attributes were changed when the model was last saved within the current request cycle. If needed, you may pass an attribute name to see if a particular attribute was changed:

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged(['title', 'slug']); // true
$user->wasChanged('first_name'); // false
$user->wasChanged(['first_name', 'title']); // true
```

The `getOriginal` method returns an array containing the original attributes of the model regardless of any changes to the model since it was retrieved. If needed, you may pass a specific attribute name to get the original value of a particular attribute:

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

$user->name; // John
$user->email; // john@example.com

$user->name = 'Jack';
$user->name; // Jack

$user->getOriginal('name'); // John
$user->getOriginal(); // Array of original attributes...
```

The `getChanges` method returns an array containing the attributes that changed when the model was last saved:

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

$user->name; // John
$user->email; // john@example.com

$user->update([
    'name' => 'Jack',
    'email' => 'jack@example.com',
]);

$user->getChanges();

/*
    [
        'name' => 'Jack',
        'email' => 'jack@example.com',
    ]
*/
```

Mass Assignment

You may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the method:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default.

A mass assignment vulnerability occurs when a user passes an unexpected HTTP request field and that field changes a column in your database that you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed to your model's `create` method, allowing the user to escalate themselves to an administrator.

So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the `$fillable` property on the model. For example, let's make the `name` attribute of our `Flight` model mass assignable:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array<int, string>
     */
    protected $fillable = ['name'];
}
```

Once you have specified which attributes are mass assignable, you may use the `create` method to insert a new record in the database. The `create` method returns the newly created model instance:

```php
$flight = Flight::create(['name' => 'London to Paris']);
```

If you already have a model instance, you may use the `fill` method to populate it with an array of attributes:

```php
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

Mass Assignment and JSON Columns

When assigning JSON columns, each column's mass assignable key must be specified in your model's `$fillable` array. For security, Laravel does not support updating nested JSON attributes when using the `guarded` property:

```php
/**
 * The attributes that are mass assignable.
 *
 * @var array<int, string>
 */
protected $fillable = [
    'options->enabled',
];
```

Allowing Mass Assignment

If you would like to make all of your attributes mass assignable, you may define your model's `$guarded` property as an empty array. If you choose to unguard your model, you should take special care to always hand-craft the arrays passed to Eloquent's `fill`, `create`, and `update` methods:

```
/**
 * The attributes that aren't mass assignable.
 *
 * @var array<string>|bool
 */
protected $guarded = [];
```

Mass Assignment Exceptions

By default, attributes that are not included in the `$fillable` array are silently discarded when performing mass-assignment operations. In production, this is expected behavior; however, during local development it can lead to confusion as to why model changes are not taking effect.

If you wish, you may instruct Laravel to throw an exception when attempting to fill an unfillable attribute by invoking the `preventSilentlyDiscardingAttributes` method. Typically, this method should be invoked in the `boot` method of your application's `AppServiceProvider` class:

```
use Illuminate\Database\Eloquent\Model;
/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Model::preventSilentlyDiscardingAttributes($this->app->isLocal());
}
```

Upserts

Eloquent's `upsert` method may be used to update or create records in a single, atomic operation. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of the columns that should be updated if a matching record already exists in the database. The `upsert` method will automatically set the `created_at` and `updated_at` timestamps if timestamps are enabled on the model:

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], uniqueBy: ['departure', 'destination'], update: ['price']);
```

> [!WARNING] All databases except SQL Server require the columns in the second argument of the `upsert` method to have a "primary" or "unique" index. In addition, the MariaDB and MySQL database drivers ignore the second argument of the `upsert` method and always use the "primary" and "unique" indexes of the table to detect existing records.

Deleting Models

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

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();
```

Deleting an Existing Model by its Primary Key

In the example above, we are retrieving the model from the database before calling the `delete` method. However, if you know the primary key of the model, you may delete the model without explicitly retrieving it by calling the `destroy` method. In addition to accepting the single primary key, the `destroy` method will accept multiple primary keys, an array of primary keys, or a collection of primary keys:

```
Flight::destroy(1);

Flight::destroy(1, 2, 3);

Flight::destroy([1, 2, 3]);
Flight::destroy(collect([1, 2, 3]));
```

If you are utilizing soft deleting models, you may permanently delete models via the `forceDestroy` method:

```
Flight::forceDestroy(1);
```

> [!WARNING] The `destroy` method loads each model individually and calls the `delete` method so that the `deleting` and `deleted` events are properly dispatched for each model.

Deleting Models Using Queries

Of course, you may build an Eloquent query to delete all models matching your query's criteria. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not dispatch model events for the models that are deleted:

```
$deleted = Flight::where('active', 0)->delete();
```

To delete all models in a table, you should execute a query without adding any conditions:

```
$deleted = Flight::query()->delete();
```

> [!WARNING] When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be dispatched for the deleted models. This is because the models are never actually retrieved when executing the delete statement.

Soft Deleting

In addition to actually removing records from your database, Eloquent can also "soft delete" models. When models are soft deleted, they are not actually removed from your database. Instead, a `deleted_at` attribute is set on the model indicating the date and time at which the model was "deleted". To enable soft deletes for a model, add the `Illuminate\Database\Eloquent\SoftDeletes` trait to the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
}
```

> [!NOTE] The `SoftDeletes` trait will automatically cast the `deleted_at` attribute to a `DateTime` / `Carbon` instance for you.

You should also add the `deleted_at` column to your database table. The Laravel schema builder contains a helper method to create this column:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. However, the model's database record will be left in the table. When querying a model that uses soft deletes, the soft deleted models will automatically be excluded from all query results.

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

```
if ($flight->trashed()) {
    // ...
}
```

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model, you may call the `restore` method on a model instance. The `restore` method will set the model's `deleted_at` column to `null`:

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

You may also use the `restore` method in a query to restore multiple models. Again, like other "mass" operations, this will not dispatch any model events for the models that are restored:

```
Flight::withTrashed()->where('airline_id', 1)->restore();
```

The `restore` method may also be used when building relationship queries:

```
$flight->history()->restore();
```

Permanently Deleting Models

Sometimes you may need to truly remove a model from your database. You may use the `forceDelete` method to permanently remove a soft deleted model from the database table:

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

You may also use the `forceDelete` method when building Eloquent relationship queries:

```
$flight->history()->forceDelete();
```

Querying Soft Deleted Models

Including Soft Deleted Models

As noted above, soft deleted models will automatically be excluded from query results. However, you may force soft deleted models to be included in a query's results by calling the `withTrashed` method on the query:

```
use App\Models\Flight;

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

The `withTrashed` method may also be called when building a relationship query:

```
$flight->history()->withTrashed()->get();
```

Retrieving Only Soft Deleted Models

The `onlyTrashed` method will retrieve **only** soft deleted models:

```
$flights = Flight::onlyTrashed()
    ->where('airline_id', 1)
    ->get();
```

Pruning Models

Sometimes you may want to periodically delete models that are no longer needed. To accomplish this, you may add the `Illuminate\Database\Eloquent\Prunable` or `Illuminate\Database\Eloquent\MassPrunable` trait to the models you would like to periodically prune. After adding one of the traits to the model, implement a `prunable` method which returns an Eloquent query builder that resolves the models that are no longer needed:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
    use Prunable;

    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

When marking models as `Prunable`, you may also define a `pruning` method on the model. This method will be called before the model is deleted. This method can be useful for deleting any additional resources associated with the model, such as stored files, before the model is permanently removed from the database:

```php
/**
 * Prepare the model for pruning.
 */
protected function pruning(): void
{
    // ...
}
```

After configuring your prunable model, you should schedule the `model:prune` Artisan command in your application's `routes/console.php` file. You are free to choose the appropriate interval at which this command should be run:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('model:prune')->daily();
```

Behind the scenes, the `model:prune` command will automatically detect "Prunable" models within your application's `app/Models` directory. If your models are in a different location, you may use the `--model` option to specify the model class names:

```
Schedule::command('model:prune', [
    '--model' => [Address::class, Flight::class],
])->daily();
```

If you wish to exclude certain models from being pruned while pruning all other detected models, you may use the `--except` option:

```
Schedule::command('model:prune', [
    '--except' => [Address::class, Flight::class],
])->daily();
```

You may test your `prunable` query by executing the `model:prune` command with the `--pretend` option. When pretending, the `model:prune` command will simply report how many records would be pruned if the command were to actually run:

```
php artisan model:prune --pretend
```

> [!WARNING] Soft deleting models will be permanently deleted (`forceDelete`) if they match the prunable query.

Mass Pruning

When models are marked with the `Illuminate\Database\Eloquent\MassPrunable` trait, models are deleted from the database using mass-deletion queries. Therefore, the `pruning` method will not be invoked, nor will the `deleting` and `deleted` model events be dispatched. This is because the models are never actually retrieved before deletion, thus making the pruning process much more efficient:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
    use MassPrunable;

    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

Replicating Models

You may create an unsaved copy of an existing model instance using the `replicate` method. This method is particularly useful when you have model instances that share many of the same attributes:

```php
use App\Models\Address;

$shipping = Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
    'state' => 'CA',
    'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
    'type' => 'billing'
]);

$billing->save();
```

To exclude one or more attributes from being replicated to the new model, you may pass an array to the `replicate` method:

```
$flight = Flight::create([
    'destination' => 'LAX',
    'origin' => 'LHR',
    'last_flown' => '2020-03-04 11:00:00',
    'last_pilot_id' => 747,
]);

$flight = $flight->replicate([
    'last_flown',
    'last_pilot_id'
]);
```

Query Scopes

Global Scopes

Global scopes allow you to add constraints to all queries for a given model. Laravel's own soft delete functionality utilizes global scopes to only retrieve "non-deleted" models from the database. Writing your own global scopes can provide a convenient, easy way to make sure every query for a given model receives certain constraints.

Generating Scopes

To generate a new global scope, you may invoke the `make:scope` Artisan command, which will place the generated scope in your application's `app/Models/Scopes` directory:

```
php artisan make:scope AncientScope
```

Writing Global Scopes

Writing a global scope is simple. First, use the `make:scope` command to generate a class that implements the `Illuminate\Database\Eloquent\Scope` interface. The `Scope` interface requires you to implement one method: `apply`. The `apply` method may add `where` constraints or other types of clauses to the query as needed:

```php
<?php

namespace App\Models\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
    /**
     * Apply the scope to a given Eloquent query builder.
     */
    public function apply(Builder $builder, Model $model): void
    {
        $builder->where('created_at', '<', now()->subYears(2000));
    }
}
```

> [!NOTE] If your global scope is adding columns to the select clause of the query, you should use the `addSelect` method instead of `select`. This will prevent the unintentional replacement of the query's existing select clause.

Applying Global Scopes

To assign a global scope to a model, you may simply place the `ScopedBy` attribute on the model:

```php
<?php

namespace App\Models;

use App\Models\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Attributes\ScopedBy;

#[ScopedBy([AncientScope::class])]
class User extends Model
{
    //
}
```

Or, you may manually register the global scope by overriding the model's `booted` method and invoke the model's `addGlobalScope` method. The `addGlobalScope` method accepts an instance of your scope as its only argument:

```
namespace App\Models;
use App\Models\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     */
    protected static function booted(): void
    {
        static::addGlobalScope(new AncientScope);
    }
}
```

After adding the scope in the example above to the `App\Models\User` model, a call to the `User::all()` method will execute the following SQL query:

```
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

Anonymous Global Scopes

Eloquent also allows you to define global scopes using closures, which is particularly useful for simple scopes that do not warrant a separate class of their own. When defining a global scope using a closure, you should provide a scope name of your own choosing as the first argument to the `addGlobalScope` method:

```
namespace App\Models;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     */
    protected static function booted(): void
    {
        static::addGlobalScope('ancient', function (Builder $builder) {
            $builder->where('created_at', '<', now()->subYears(2000));
        });
    }
}
```

Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. This method accepts the class name of the global scope as its only argument:

```
User::withoutGlobalScope(AncientScope::class)->get();
```

Or, if you defined the global scope using a closure, you should pass the string name that you assigned to the global scope:

```
User::withoutGlobalScope('ancient')->get();
```

If you would like to remove several or even all of the query's global scopes, you may use the `withoutGlobalScopes` method:

```
// Remove all of the global scopes...
User::withoutGlobalScopes()->get();

// Remove some of the global scopes...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

Local Scopes

Local scopes allow you to define common sets of query constraints that you may easily re-use throughout your application. For example, you may need to frequently retrieve all users that are considered "popular". To define a scope, prefix an Eloquent model method with `scope`.

Scopes should always return the same query builder instance or `void`:

```php
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     */
    public function scopePopular(Builder $query): void
    {
        $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     */
    public function scopeActive(Builder $query): void
    {
        $query->where('active', 1);
    }
}
```

Utilizing a Local Scope

Once the scope has been defined, you may call the scope methods when querying the model. However, you should not include the `scope` prefix when calling the method. You can even chain calls to various scopes:

```php
use App\Models\User;
$users = User::popular()->active()->orderBy('created_at')->get();
```

Combining multiple Eloquent model scopes via an `or` query operator may require the use of closures to achieve the correct logical grouping:

```php
$users = User::popular()->orWhere(function (Builder $query) {
    $query->active();
})->get();
```

However, since this can be cumbersome, Laravel provides a "higher order" `orWhere` method that allows you to fluently chain scopes together without the use of closures:

```php
$users = User::popular()->orWhere->active()->get();
```

Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. To get started, just add your additional parameters to your scope method's signature. Scope parameters should be defined after the `$query` parameter:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include users of a given type.
     */
    public function scopeOfType(Builder $query, string $type): void
    {
        $query->where('type', $type);
    }
}
```

Once the expected arguments have been added to your scope method's signature, you may pass the arguments when calling the scope:

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

Pending Attributes

If you would like to use scopes to create models that have the same attributes as those used to constrain the scope, you may use the `withAttributes` method when building the scope query:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Scope the query to only include drafts.
     */
    public function scopeDraft(Builder $query): void
    {
        $query->withAttributes([
            'hidden' => true,
        ]);
    }
}
```

The `withAttributes` method will add `where` conditions to the query using the given attributes, and it will also add the given attributes to any models created via the scope:

```php
$draft = Post::draft()->create(['title' => 'In Progress']);

$draft->hidden; // true
```

To instruct the `withAttributes` method to not add `where` conditions to the query, you may set the `asConditions` argument to `false`:

```php
$query->withAttributes([
    'hidden' => true,
], asConditions: false);
```

Comparing Models

Sometimes you may need to determine if two models are the "same" or not. The `is` and `isNot` methods may be used to quickly verify two models have the same primary key, table, and database connection or not:

```
if ($post->is($anotherPost)) {
    // ...
}

if ($post->isNot($anotherPost)) {
    // ...
}
```

The `is` and `isNot` methods are also available when using the `belongsTo`, `hasOne`, `morphTo`, and `morphOne` relationships. This method is particularly helpful when you would like to compare a related model without issuing a query to retrieve that model:

```
if ($post->author()->is($user)) {
    // ...
}
```

Events

> [!NOTE] Want to broadcast your Eloquent events directly to your client-side application? Check out Laravel's model event broadcasting.

Eloquent models dispatch several events, allowing you to hook into the following moments in a model's lifecycle: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `trashed`, `forceDeleting`, `forceDeleted`, `restoring`, `restored`, and `replicating`.

The `retrieved` event will dispatch when an existing model is retrieved from the database. When a new model is saved for the first time, the `creating` and `created` events will dispatch. The `updating` / `updated` events will dispatch when an existing model is modified and the `save` method is called. The `saving` / `saved` events will dispatch when a model is created or updated - even if the model's attributes have not been changed. Event names ending with `-ing` are dispatched before any changes to the model are persisted, while events ending with `-ed` are dispatched after the changes to the model are persisted.

To start listening to model events, define a `$dispatchesEvents` property on your Eloquent model. This property maps various points of the Eloquent model's lifecycle to your own event classes. Each model event class should expect to receive an instance of the affected model via its constructor:

```
namespace App\Models;
use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
    /**
     * The event map for the model.
     * @var array<string, string>
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

After defining and mapping your Eloquent events, you may use event listeners to handle the events.

> [!WARNING] When issuing a mass update or delete query via Eloquent, the `saved`, `updated`, `deleting`, and `deleted` model events will not be dispatched for the affected models. This is because the models are never actually retrieved when performing mass updates or deletes.

Using Closures

Instead of using custom event classes, you may register closures that execute when various model events are dispatched. Typically, you should register these closures in the `booted` method of your model:

```
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{
    /**
     * The "booted" method of the model.
     */
    protected static function booted(): void
    {
        static::created(function (User $user) {
            // ...
        });
    }
}
```

If needed, you may utilize queueable anonymous event listeners when registering model events. This will instruct Laravel to execute the model event listener in the background using your application's queue:

```
use function Illuminate\Events\queueable;

static::created(queueable(function (User $user) {
    // ...
}));
```

Observers

Defining Observers

If you are listening for many events on a given model, you may use observers to group all of your listeners into a single class. Observer classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the affected model as their only argument. The `make:observer` Artisan command is the easiest way to create a new observer class:

```
php artisan make:observer UserObserver --model=User
```

This command will place the new observer in your `app/Observers` directory. If this directory does not exist, Artisan will create it for you. Your fresh observer will look like the following:

```php
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Handle the User "created" event.
     */
    public function created(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "updated" event.
     */
    public function updated(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "deleted" event.
     */
    public function deleted(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "restored" event.
     */
    public function restored(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "forceDeleted" event.
     */
    public function forceDeleted(User $user): void
    {
        // ...
    }
}
```

To register an observer, you may place the `ObservedBy` attribute on the corresponding model:

```
use App\Observers\UserObserver;
use Illuminate\Database\Eloquent\Attributes\ObservedBy;

#[ObservedBy([UserObserver::class])]
class User extends Authenticatable
{
    //
}
```

Or, you may manually register an observer by invoking the `observe` method on the model you wish to observe. You may register observers in the `boot` method of your application's `AppServiceProvider` class:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    User::observe(UserObserver::class);
}
```

> [!NOTE] There are additional events an observer can listen to, such as `saving` and `retrieved`. These events are described within the events documentation.

Observers and Database Transactions

When models are being created within a database transaction, you may want to instruct an observer to only execute its event handlers after the database transaction is committed. You may accomplish this by implementing the `ShouldHandleEventsAfterCommit` interface on your observer. If a database transaction is not in progress, the event handlers will execute immediately:

```php
<?php

namespace App\Observers;

use App\Models\User;
use Illuminate\Contracts\Events\ShouldHandleEventsAfterCommit;

class UserObserver implements ShouldHandleEventsAfterCommit
{
    /**
     * Handle the User "created" event.
     */
    public function created(User $user): void
    {
        // ...
    }
}
```

Muting Events

You may occasionally need to temporarily "mute" all events fired by a model. You may achieve this using the `withoutEvents` method. The `withoutEvents` method accepts a closure as its only argument. Any code executed within this closure will not dispatch model events, and any value returned by the closure will be returned by the `withoutEvents` method:

```php
use App\Models\User;

$user = User::withoutEvents(function () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

Saving a Single Model Without Events

Sometimes you may wish to "save" a given model without dispatching any events. You may accomplish this using the `saveQuietly` method:

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

$user->name = 'Victoria Faith';

$user->saveQuietly();
```

You may also "update", "delete", "soft delete", "restore", and "replicate" a given model without dispatching any events:

```
$user->deleteQuietly();
$user->forceDeleteQuietly();
$user->restoreQuietly();
```