VERSION

5.8

Search Docs

Eloquent: Getting Started

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. Models allow you to query for data in your tables, as well as insert new records into the table.

Before getting started, be sure to configure a database connection in `config/database.php`. For more information on configuring your database, check out the documentation.

Defining Models

To get started, let's 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` class.

The easiest way to create a model instance is using the `make:model` Artisan command:

```
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

php artisan make:model Flight -m
```

Eloquent Model Conventions

Now, let's look at an example `Flight` model, which we will use to retrieve and store information from our `flights` database table:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

Table Names

Note that we did not tell Eloquent which table to use for 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. You may specify a custom table by defining a `table` property on your model:

```php
<?php

namespace App;

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 table has a primary key column named `id`. You may define a protected `$primaryKey` property to override this convention:

```
<?php

namespace App;

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 by default the primary key will automatically be cast to an `int`. If you wish to use a non-incrementing or a non-numeric primary key you must set the public `$incrementing` property on your model to `false`:

```
<?php

class Flight extends Model
{
    /**
```

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

If your primary key is not an integer, you should set the protected `$keyType` property on your model to `string`:

```
<?php

class Flight extends Model
{
    /**
     * The "type" of the auto-incrementing ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the `$timestamps` property on your model to `false`:

```
<?php

namespace App;

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 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

namespace App;

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 set the `CREATED_AT` and `UPDATED_AT` constants in your model:

```
<?php
```

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

Database Connection

By default, all Eloquent models will use the default database connection configured for your application. If you would like to specify a different connection for the model, use the `$connection` property:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The connection name for the model.
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

Default 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:

```
<?php

namespace App;
```

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

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

Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example:

```
<?php

$flights = App\Flight::all();

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

Adding Additional Constraints

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

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

Since Eloquent models are query builders, you should review all of the methods available on the query builder. You may use any of these methods in your Eloquent queries.

Refreshing Models

You can refresh models 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 = App\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 = App\Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';
```

```
$flight->refresh();

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

Collections

For Eloquent methods like `all` and `get` which retrieve multiple results, an instance of `Illuminate\Database\Eloquent\Collection` will be returned. The `Collection` class provides a variety of helpful methods for working with your Eloquent results:

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

You may also loop over the collection like an array:

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

Chunking Results

If you need to process thousands of Eloquent records, use the `chunk` command. The `chunk` method will retrieve a "chunk" of Eloquent models, feeding them to a given `Closure` for processing. Using the `chunk` method will conserve memory when working with large result sets:

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

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 retrieved from the database. A database query will be executed to retrieve each chunk of records passed to the Closure.

Using Cursors

The `cursor` method allows you to iterate through your database records using a cursor, which will only execute a single query. When processing large amounts of data, the `cursor` method may be used to greatly reduce your memory usage:

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) {
    //
}
```

Retrieving Single Models / Aggregates

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

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

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

You may also call the `find` method with an array of primary keys, which will return a collection of the matching records:

```
$flights = App\Flight::find([1, 2, 3]);
```

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, a `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

If the exception is not caught, a `404` HTTP response is automatically sent back to the user. It is not necessary to write explicit checks to return `404` responses when using these methods:

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

Retrieving Aggregates

You may also use the `count`, `sum`, `max`, and other aggregate methods provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

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

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

Inserting & Updating Models

Inserts

To create a new record in the database, create a new model instance, set attributes on the model, then call the `save` method:

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class FlightController extends Controller
{
    /**
     * Create a new flight instance.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;

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

        $flight->save();
    }
}
```

In this example, we assign the `name` parameter from the incoming HTTP request to the `name` attribute of the `App\Flight` model instance. When we call

the `save` method, a record will be inserted into the database. The `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.

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, set any attributes you wish to update, and then call the `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();
```

Mass Updates

Updates can also be performed against any number of 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:

```
App\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.

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.

Mass Assignment

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 by default.

A mass-assignment vulnerability occurs when a user passes an unexpected HTTP parameter through a request, and that parameter changes a column in your database you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed into 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

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
```

```
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once we have made the attributes mass assignable, we can use the `create` method to insert a new record in the database. The `create` method returns the saved model instance:

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

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

```
$flight->fill(['name' => 'Flight 22']);
```

Guarding Attributes

While `$fillable` serves as a "white list" of attributes that should be mass assignable, you may also choose to use `$guarded`. The `$guarded` property should contain an array of attributes that you do not want to be mass assignable. All other attributes not in the array will be mass assignable. So, `$guarded` functions like a "black list". Importantly, you should use either `$fillable` or `$guarded` - not both. In the example below, all attributes **except for** `price` will be mass assignable:

```
<?php

namespace App;
```

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

class Flight extends Model
{
    /**
     * The attributes that aren't mass assignable.
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

If you would like to make all attributes mass assignable, you may define the `$guarded` property as an empty array:

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

Other Creation Methods

`firstOrCreate` / `firstOrNew`

There are two other methods you may use to create models by mass assigning attributes: `firstOrCreate` and `firstOrNew`. The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model can not be found in the database, a record will be inserted with the attributes from the first parameter, along with those in the optional second parameter.

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 call `save` manually to persist it:

```
// Retrieve flight by name, or create it if it doesn't exist...
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

// Retrieve flight by name, or create it with the name, delayed, and arrival_
$flight = App\Flight::firstOrCreate(
    ['name' => 'Flight 10'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Retrieve by name, or instantiate...
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

// Retrieve by name, or instantiate with the name, delayed, and arrival_time
$flight = App\Flight::firstOrNew(
    ['name' => 'Flight 10'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

`updateOrCreate`

You may also come across situations where you want to update an existing model or create a new model if none exists. Laravel provides an `updateOrCreate` method to do this in one step. Like the `firstOrCreate` method, `updateOrCreate` persists the model, so there's no need to call `save()`:

```
// If there's a flight from Oakland to San Diego, set the price to $99.
// If no matching model exists, create one.
$flight = App\Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

Deleting Models

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

```
$flight = App\Flight::find(1);

$flight->delete();
```

Deleting An Existing Model By 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 retrieving it by calling the `destroy` method. In addition to a single primary key as its argument, the `destroy` method will accept multiple primary keys, an array of primary keys, or a collection of primary keys:

```
App\Flight::destroy(1);

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

App\Flight::destroy([1, 2, 3]);

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

Deleting Models By Query

You can also run a delete statement on a set of models. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not fire any model events for the models that are deleted:

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

When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be fired 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 and inserted into the database. If a model has a non-null `deleted_at` value, the model has been soft deleted. To enable soft deletes for a model, use the `Illuminate\Database\Eloquent\SoftDeletes` trait on the model:

```
<?php

namespace App;

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

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

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:

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

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. And, 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, use the `trashed` method:

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

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 appear in a result set using the `withTrashed` method on the query:

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

The `withTrashed` method may also be used on a relationship query:

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

Retrieving Only Soft Deleted Models

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

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

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model into an active state, use the `restore` method on a model instance:

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

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

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

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

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

Permanently Deleting Models

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

```
// Force deleting a single model instance...
$flight->forceDelete();

// Force deleting all related models...
$flight->history()->forceDelete();
```

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 pull "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.

Writing Global Scopes

Writing a global scope is simple. Define a class that implements the `Illuminate\Database\Eloquent\Scope` interface. This interface requires you to

implement one method: `apply`. The `apply` method may add `where` constraints to the query as needed:

```php
<?php

namespace App\Scopes;

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

class AgeScope implements Scope
{
    /**
     * 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->where('age', '>', 200);
    }
}
```

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 should override a given model's `boot` method and use the `addGlobalScope` method:

```php
<?php

namespace App;

use App\Scopes\AgeScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope(new AgeScope);
    }
}
```

After adding the scope, a query to `User::all()` will produce the following SQL:

```
select * from `users` where `age` > 200
```

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:

```php
<?php

namespace App;

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

class User extends Model
{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('age', function (Builder $builder) {
            $builder->where('age', '>', 200);
        });
    }
}
```

Removing Global Scopes

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

```php
User::withoutGlobalScope(AgeScope::class)->get();
```

Or, if you defined the global scope using a Closure:

```php
User::withoutGlobalScope('age')->get();
```

If you would like to remove several or even all of the 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 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 a query builder instance:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
```

```
        return $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeActive($query)
    {
        return $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, for example:

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

Combining multiple Eloquent model scopes via an `or` query operator may require the use of Closure callbacks:

```
$users = App\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 these scopes together without the use of Closures:

```
$users = App\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. Scope parameters should be defined after the `$query` parameter:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include users of a given type.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @param  mixed  $type
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

Now, you may pass the parameters when calling the scope:

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

Comparing Models

Sometimes you may need to determine if two models are the "same". The `is` method may be used to quickly verify two models have same primary key, table, and database connection:

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

Events

Eloquent models fire several events, allowing you to hook into the following points in a model's lifecycle: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`. Events allow you to easily execute code each time a specific model class is saved or updated in the database. Each event receives the instance of the model through its constructor.

The `retrieved` event will fire 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 fire. If a model already existed in the database and the `save` method is called, the `updating` / `updated` events will fire. However, in both cases, the `saving` / `saved` events will fire.

When issuing a mass update via Eloquent, the `saved` 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.

To get started, define a `$dispatchesEvents` property on your Eloquent model that maps various points of the Eloquent model's lifecycle to your own event classes:

```php
<?php

namespace App;

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

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

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

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. Observers classes have method

names which reflect the Eloquent events you wish to listen for. Each of these methods receives the 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

namespace App\Observers;

use App\User;

class UserObserver
{
    /**
     * Handle the User "created" event.
     *
     * @param  \App\User  $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * Handle the User "updated" event.
     *
     * @param  \App\User  $user
     * @return void
     */
    public function updated(User $user)
```

```
    {
        //
    }

    /**
     * Handle the User "deleted" event.
     *
     * @param  \App\User  $user
     * @return void
     */
    public function deleted(User $user)
    {
        //
    }
}
```

To register an observer, use the `observe` method on the model you wish to observe. You may register observers in the `boot` method of one of your service providers. In this example, we'll register the observer in the `AppServiceProvider`:

```
<?php

namespace App\Providers;

use App\User;
use App\Observers\UserObserver;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
```

```
    }

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

Partners +

Ecosystem +

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