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.
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

```
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
```

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

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{
    //
}
```

**Table Names**

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.

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
```

```
{
    protected $table = 'my_flights';
}
```

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
{
    protected $primaryKey = 'flight_id';
}
```

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
{
    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
{
    protected $keyType = 'string';
}
```

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:

```
public $timestamps = false;
```

If you need to customize the format of your timestamps, set the $dateFormat property on your model.

```
    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:

```
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

```
    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

```
protected $attributes = [
        'delayed' => false,
    ];
```

**Retrieving Models**

Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model.

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

The Eloquent all method will return all of the results in the model's table.

Use the get method to retrieve the results:

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

```
            ->take(10)
            ->get();
```

**Refreshing Models**

- refresh() is a mutable operation: It will reload the current model instance from the database.
- fresh() is an immutable operation: It returns a new model instance from the database. It doesn't affect the current instance.

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

$flight = App\Flight::where('number', 'FR 900')->first();
$flight->number = 'FR 456';
$flight->refresh();
$flight->number; // "FR 900"
```

Subquery Selects

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

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

```
// 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();

// Shorthand for retrieving the first model matching the query constraints...
$flight = App\Flight::firstWhere('active', 1);
```

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]);
```

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);
});
```

Aggregates

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

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

```
$flight = new Flight;
$flight->name = $request->name;
$flight->save();
```

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.

Update:

```
$flight = App\Flight::find(1);
$flight->name = 'New Flight Name';
$flight->save();
```

Updates can also be performed against any number of models that match a given query.

```
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.

**Examining Attribute Changes**

The isDirty method determines if any attributes have been changed since the model was loaded. You may pass a specific attribute name to determine if a particular attribute is dirty. The isClean method is the opposite of isDirty and also accepts an optional attribute argument:

```
$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->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true

$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.

```
$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('first_name'); // false
```

The getOriginal method returns an array containing the original attributes of the model regardless of any changes since the model was loaded.

```
$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...
```

**Mass Assignment**

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.

```
protected $fillable = ['name']; // Model


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


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

While $fillable serves as a "allow 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.

So, $guarded functions like a "deny list"

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

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 cannot 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_time attributes...
$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 attributes...
$flight = App\Flight::firstOrNew(
    ['name' => 'Flight 10'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

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]
);
```

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

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

$flight->delete();
```

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 a single primary key as its argument, the destroy method will accept multiple 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

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

**Soft Deleting**

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.

```
<?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.

```
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.

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

The onlyTrashed method will retrieve only soft deleted models:

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

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();
```

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();
```

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

```
$shipping = App\Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
```

```
    'state' => 'CA',
    'postcode' => '90001',
]);

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

$billing->save();
```

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.