Eloquent ORM:

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.

يتضمن الـ Laravel الـ Eloquent، وهو برنامج تعيين كائنات العلاقات (ORM) يجعل التفاعل مع قاعدة البيانات أمرًا ممتعًا. عند استخدام Eloquent، يكون لكل جدول في قاعدة البيانات "نموذج" مطابق يستخدم للتفاعل مع هذا الجدول.
بالإضافة إلى استرجاع السجلات من جدول قاعدة البيانات، تتيح لك نماذج Eloquent اضافة السجلات وتحديثها وحذفها من الجدول أيضًا.

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:

للبدء، دعنا ننشئ نموذج Eloquent. توجد النماذج عادةً في دليل app\Models وترث الفئة Illuminate\Database\Eloquent\Model.
يمكنك استخدام أمر make:model Artisan لإنشاء نموذج جديد:

```
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:

إذا كنت ترغب في انشاء جدول قاعدة البيانات عند إنشاء الـ model، فيمكنك استخدام الخيار migration-- أو m- :

```
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:

يمكنك إنشاء أنواع أخرى مختلفة من الفئات عند إنشاء نموذج، مثل الـ factories، الـ seeders، الـ policies، الـ controllers، والـ form requests.
بالإضافة إلى ذلك، يمكن دمج هذه الخيارات لإنشاء فئات متعددة في وقت واحد:

```
# 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:

في بعض الأحيان قد يكون من الصعب تحديد جميع السمات والعلاقات المتاحة للنموذج بمجرد قراءة الكود الخاص به. بدلاً من ذلك، استخدم الأمر model:show، الذي يوفر نظرة عامة ملائمة على جميع سمات النموذج وعلاقاته:

```
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:

سيتم وضع النماذج التي تم إنشاؤها بواسطة الأمر make:model في دليل app/Models. دعنا نفحص فئة نموذج أساسية ونناقش بعض مفاهيم الـ Eloquent الرئيسية:

```
<?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.

بعد إلقاء نظرة سريعة على المثال أعلاه، ربما لاحظت أننا لم نخبر Eloquent بوجود اي جدول في قاعدة البيانات يتوافق مع نموذج Flight الخاص بنا. وفقًا للمبدأ، سيتم استخدام "حالة الثعبان للتسمية"، اسم الجمع للفئة كاسم الجدول ما لم يتم تحديد اسم آخر صراحةً. لذا، في هذه الحالة، سيفترض Eloquent أن نموذج Flight يخزن السجلات في جدول flights، بينما يخزن نموذج AirTrafficController السجلات في جدول air_traffic_controllers.

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:

إذا كان جدول قاعدة البيانات المقابل للنموذج الخاص بك لا يتوافق مع هذه المفهوم، فيمكنك تحديد اسم جدول النموذج يدويًا عن طريق تعريف خاصية الجدول في النموذج:

```
<?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:

سيفترض Eloquent أيضًا أن جدول قاعدة البيانات المقابل لكل نموذج يحتوي على عمود مفتاح أساسي يسمى id. إذا لزم الأمر، يمكنك تعريف خاصية $primaryKey المحمية على النموذج الخاص بك لتحديد عمود مختلف يعمل كمفتاح أساسي للنموذج الخاص بك:

```
<?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:

بالإضافة إلى ذلك، يفترض Eloquent أن المفتاح الأساسي عبارة عن قيمة عدد صحيح متزايدة، مما يعني أن Eloquent سيقوم تلقائيًا بتحويل المفتاح الأساسي إلى عدد صحيح. إذا كنت ترغب في استخدام مفتاح أساسي غير متزايد أو غير رقمي، فيجب عليك تعريف خاصية $incrementing عامة على النموذج الخاص بك والتي تم تعيينها على false:

```
<?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:

إذا لم يكن المفتاح الأساسي للنموذج الخاص بك عددًا صحيحًا، فيجب عليك تعريف خاصية $keyType محمية على النموذج الخاص بك. يجب أن تكون قيمة هذه الخاصية سلسلة:

```
<?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.

يتطلب Eloquent أن يكون لكل نموذج "معرف" فريد واحد على الأقل يمكن أن يعمل كمفتاح أساسي له. لا تدعم نماذج Eloquent المفاتيح الأساسية "المركبة". ومع ذلك، يمكنك إضافة فهارس فريدة متعددة الأعمدة إضافية إلى جداول قاعدة البيانات الخاصة بك بالإضافة إلى المفتاح الأساسي الفريد للجدول.

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.

بدلاً من استخدام الأعداد الصحيحة المتزايدة تلقائيًا كمفاتيح أساسية لنموذج Eloquent الخاص بك، يمكنك اختيار استخدام معرفات UUID بدلاً من ذلك. معرفات UUID هي معرفات أبجدية رقمية عالمية فريدة يبلغ طولها 36 حرفًا.

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:

إذا كنت ترغب في أن يستخدم النموذج مفتاح UUID بدلاً من مفتاح عدد صحيح متزايد تلقائيًا، فيمكنك استخدام السمة Illuminate\Database\Eloquent\Concerns\HasUuids على النموذج. بالطبع، يجب عليك التأكد من أن النموذج يحتوي على عمود مفتاح أساسي مكافئ لـ UUID:

```
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.

بشكل افتراضي، ستقوم السمة HasUuids بإنشاء معرفات UUID "مرتبة" لنماذجك. تعد هذه المعرفات أكثر كفاءة لتخزين قاعدة البيانات المفهرسة لأنها يمكن فهرستها بنفس طريقة فهرسة المعاجم.

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:

يمكنك تجاوز عملية إنشاء UUID لنموذج معين من خلال تحديد الدالة newUniqueId على النموذج. بالإضافة إلى ذلك، يمكنك تحديد الأعمدة التي يجب أن تتلقى UUIDs من خلال تحديد الدالة uniqueIds على النموذج:

```
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:

إذا كنت ترغب في استخدام "معرفات ULID" بدلاً من معرفات UUID. معرفات ULID تشبه معرفات UUID؛ ومع ذلك، فهي تتكون من 26 حرفًا فقط. ومثل معرفات UUID المرتبة، يمكن فرز معرفات ULID معجميًا لفهرسة قاعدة البيانات بكفاءة. لاستخدام معرفات ULID، يجب عليك استخدام السمة Illuminate\Database\Eloquent\Concerns\HasUlids في نموذجك. يجب عليك أيضًا التأكد من أن النموذج يحتوي على عمود مفتاح أساسي مكافئ لمعرف ULID:

```
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:

بشكل افتراضي، يتوقع Eloquent وجود العمودين created_at وupdate_at في جدول قاعدة البيانات المقابل للنموذج.
سيقوم Eloquent تلقائيًا بتعيين قيم هذه الأعمدة عند إنشاء النماذج أو تحديثها.
إذا كنت لا تريد إدارة هذه الأعمدة تلقائيًا بواسطة Eloquent، فيجب عليك تعريف خاصية $timestamps في النموذج الخاص بك بقيمة 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:

إذا كنت بحاجة إلى تخصيص تنسيق الطوابع الزمنية لنموذجك، فقم بتعيين الخاصية $dateFormat على نموذجك. تحدد هذه الخاصية كيفية تخزين سمات التاريخ في قاعدة البيانات بالإضافة إلى تنسيقها عند تسلسل النموذج إلى مصفوفة أو JSON:

```
<?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:

إذا كنت بحاجة إلى تخصيص أسماء الأعمدة المستخدمة لتخزين الطوابع الزمنية، فيمكنك تعريف الثوابت CREATED_AT وUPDATED_AT على النموذج الخاص بك:

```
<?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:

إذا كنت ترغب في إجراء عمليات نموذجية دون تعديل الطابع الزمني updated_at للنموذج، فيمكنك العمل على النموذج داخل closure معين للدالة withoutTimestamps:

```
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:

بشكل افتراضي، ستستخدم جميع نماذج Eloquent اتصال قاعدة البيانات الافتراضي الذي تم تكوينه لتطبيقك. إذا كنت ترغب في تحديد اتصال مختلف يجب استخدامه عند التفاعل مع نموذج معين، فيجب عليك تعريف خاصية $connection على النموذج:

```
<?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:

افتراضيًا، لن تحتوي كائنات النموذج التي تم إنشاؤها حديثًا على أي قيم للصفات. إذا كنت ترغب في تحديد القيم الافتراضية لبعض صفات النموذج، فيمكنك تحديد خاصية $attributes على النموذج. يجب أن تكون قيم السمات الموضوعة في مجموعة $attributes بتنسيقها الخام "القابل للتخزين" كما لو كانت مقروءة للتو من قاعدة البيانات:

```
<?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:
يقدم Laravel عدة دوال تسمح لك بتكوين او تهيئة سلوك Eloquent في مجموعة متنوعة من الحالات.
أولاً، تقبل الدالة PreventLazyLoading وسيطة منطقية اختيارية تشير إلى ما إذا كان يجب منع التحميل الكسول. على سبيل المثال، قد ترغب في تعطيل التحميل الكسول فقط في البيئات غير الإنتاجية حتى تستمر بيئة الإنتاج الخاصة بك في العمل بشكل طبيعي حتى إذا كانت علاقة التحميل الكسول موجودة عن دالة الخطأ في كود الإنتاج. عادةً، يجب استدعاء هذه الدالة في دالة التمهيد لـ AppServiceProvider الخاص بتطبيقك:

```
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:
يمكنك أيضًا توجيه Laravel لإلقاء استثناء عند محاولة ملء سمة غير قابلة للتعبئة من خلال استدعاء الدالة PreventSilentlyDiscardingAttributes.
يمكن أن يساعد هذا في منع الأخطاء غير المتوقعة أثناء التطوير المحلي عند محاولة تعيين سمة لم تتم إضافتها إلى مجموعة النماذج القابلة للتعبئة:

```
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:
بمجرد إنشاء نموذج وجدول قاعدة البيانات المرتبط به، تصبح جاهزًا لبدء استرداد البيانات من قاعدة البيانات الخاصة بك. يمكنك التفكير في كل نموذج Eloquent باعتباره أداة بناء استعلامات قوية تسمح لك بالاستعلام بسلاسة عن جدول قاعدة البيانات المرتبط بالنموذج. ستقوم الدالة all الخاصة بالنموذج باسترداد جميع السجلات من جدول قاعدة البيانات المرتبط بالنموذج:

```
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:

ستعيد الدالة all التابعة للـ Eloquent جميع النتائج الموجودة في جدول النموذج. ونظرًا لأن كل نموذج Eloquent يعمل كمنشئ للاستعلامات، فيمكنك إضافة قيود إضافية إلى الاستعلامات ثم استدعاء الدالة get لاسترداد النتائج:

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

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:

إذا كان لديك بالفعل كائن لنموذج Eloquent تم استرجاعه من قاعدة البيانات، فيمكنك "تحديث" النموذج باستخدام الدالتين fresh و refresh.

ستعيد الدالة fresh النموذج من قاعدة البيانات. لن يتأثر مثيل النموذج الحالي:

```
$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:

ستعمل الدالة refreshعلى إعادة من قاعدة البيانات وتغذية النموذج الحالي باستخدام بيانات جديدة. بالإضافة إلى ذلك، سيتم تحديث جميع العلاقات المحملة أيضًا:

```
$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.

كما رأينا، فإن دوال Eloquent مثلها كمثل جميع الدوال الأخرى تسترد سجلات متعددة من قاعدة البيانات. ومع ذلك، لا تقوم هذه الدوال بإرجاع مصفوفة PHP عادية. بدلاً من ذلك، يتم إرجاع مثيل Illuminate\Database\Eloquent\Collection.

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:

ترث فئة Eloquent Collection الفئة Illuminate\Support\Collection الأساسية في Laravel، والتي توفر مجموعة متنوعة من الدوال المفيدة للتفاعل مع مجموعات البيانات. على سبيل المثال، يمكن استخدام الدالة reject لإزالة النماذج من مجموعة بناءً على نتائج الإغلاق المستدعى:

```
$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.

بالإضافة إلى الدوال التي توفرها فئة المجموعة الأساسية في Laravel، توفر فئة المجموعة Eloquent بعض الدوال الإضافية المخصصة للتفاعل مع مجموعات نماذج Eloquent.

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

نظرًا لأن جميع مجموعات Laravel تنفذ واجهات PHP القابلة للتكرار، فيمكنك التكرار عبر المجموعات كما لو كانت مصفوفة:

```
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.

قد ينفد مخزون الذاكرة في تطبيقك إذا حاولت تحميل عشرات الآلاف من سجلات Eloquent عبر الدالتين all أو get.
وبدلاً من استخدام هذه الدوال، يمكن استخدام الدالة chunk لمعالجة أعداد كبيرة من النماذج بكفاءة أكبر.

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:

ستقوم الدالة chunk باسترداد مجموعة فرعية من نماذج Eloquent، وتمريرها إلى الـ closure للمعالجة.
نظرًا لأنه يتم استرداد الجزء الحالي فقط من نماذج Eloquent في كل مرة، فإن الدالة chunk ستوفر استخدامًا منخفضًا للذاكرة بشكل كبير عند العمل مع عدد كبير من النماذج:

```
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.

المعامل الأول الذ يتم تمريره إلى الدالة chunk هي عدد السجلات التي ترغب في تلقيها لكل "chunk".
سيتم استدعاء الـ closure الذي تم تمريره كمعامل ثاني لكل "chunk" يتم استرداده من قاعدة البيانات.
سيتم تنفيذ استعلام قاعدة البيانات لاسترداد كل "chunk" من السجلات التي تم تمريرها إلى الـclosure .

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:

إذا كنت تقوم بتصفية نتائج الدالة chunk استنادًا إلى عمود ستقوم أيضًا بتحديثه أثناء تكرار النتائج، لذا يجب عليك استخدام الدالة chunkById.
قد يؤدي استخدام الدالة chunk في هذه السيناريوهات إلى نتائج غير متوقعة وغير متسقة.
داخليًا، ستقوم الدالة chunkById دائمًا باسترداد النماذج التي تحتوي على عمود معرف أكبر من آخر نموذج في الـ chunk السابق:

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

تعمل الدالة lazy بشكل مشابه للدالة chunk بمعنى أنها تنفذ الاستعلام في أجزاء خلف الكواليس.
ومع ذلك، بدلاً من تمرير كل جزء مباشرةً إلى الـ callback كما هو، تقوم الدالة lazy بإرجاع LazyCollection يحتوي على بيانات من نماذج Eloquent، مما يتيح لك التفاعل مع النتائج كدفق واحد:

```
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:

إذا كنت تقوم بتصفية نتائج الدالة lazy اعتمادا على عمود ستقوم أيضًا بتحديثه أثناء تكرار النتائج، لذا يجب عليك استخدام الدالة lazyById.
داخليًا، ستقوم الدالة lazyById دائمًا باسترداد النماذج التي تحتوي على عمود معرف أكبر من آخر نموذج في الجزء السابق:

```
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.

بإمكانك تصفية النتائج بناءً على الترتيب التنازلي للمعرف باستخدام الدالة lazyByIdDesc.

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.

على غرار الدالة lazy، يمكن استخدام الدالة cursor لتقليل استهلاك تطبيقك للذاكرة بشكل كبير عند التكرار عبر عشرات الآلاف من سجلات نموذج Eloquent.

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.

ستنفذ الدالة cursor استعلام قاعدة بيانات واحد فقط؛ ومع ذلك، لن يتم تغذية نماذج Eloquent الفردية حتى يتم تكرارها فعليًا.
وبالتالي، يتم الاحتفاظ بنموذج Eloquent واحد فقط في الذاكرة في أي وقت معين أثناء التكرار اعتمادا على المؤشر.

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.

نظرًا لأن الدالة cursor لا تحتفظ إلا بنموذج Eloquent واحد في الذاكرة في كل مرة، فلا يمكنها إنشاء علاقات تحميل متتالية.
إذا كنت بحاجة إلى إنشاء علاقات تحميل متتالية، ففكر في استخدام طريقة lazy بدلاً من ذلك.

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

داخليًا، تستخدم الدالة cursor مولدات PHP لتنفيذ هذه الوظيفة:

```
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:

يعيد المؤشر كائن للـ Illuminate\Support\LazyCollection.
تتيح لك الفئة LazyCollection استخدام العديد من دوال التجميع المتوفرة في مجموعات Laravel النموذجية مع تحميل نموذج واحد فقط في الذاكرة في كل مرة:

```
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.

على الرغم من أن الدالة cursor تستخدم ذاكرة أقل بكثير من الاستعلام العادي (عن طريق الاحتفاظ بنموذج Eloquent واحد فقط في الذاكرة في كل مرة)، إلا أنها ستستنفد الذاكرة في النهاية.
ويرجع هذا إلى أن برنامج تشغيل PDO الخاص بـ PHP يخزن داخليًا جميع نتائج الاستعلام الخام في المخزن المؤقت الخاص به.
إذا كنت تتعامل مع عدد كبير جدًا من سجلات Eloquent، ففكر في استخدام الدالة lazy بدلاً من ذلك.

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.

يوفر Eloquent دعمًا متقدمًا للاستعلامات الفرعية، مما يسمح لك بسحب المعلومات من الجداول ذات الصلة في استعلام واحد.
على سبيل المثال، لنتخيل أن لدينا جدول وجهات الرحلات وجدول الرحلات إلى الوجهات.
يحتوي جدول الرحلات على عمود reaching_at الذي يشير إلى وقت وصول الرحلة إلى الوجهة.

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:

باستخدام دالة الاستعلام الفرعي المتاحة للدوال select وaddSelect الخاصة بمنشئ الاستعلام، يمكننا تحديد جميع الوجهات واسم الرحلة التي وصلت مؤخرًا إلى تلك الوجهة باستخدام استعلام واحد:

```
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:

بالإضافة إلى ذلك، تدعم الدالة orderBy في منشئ الاستعلامات الاستعلامات الفرعية.
في مثال الرحلة، يمكننا استخدام هذه الوظيفة لفرز جميع الوجهات بناءً على وقت وصول آخر رحلة إلى تلك الوجهة. مرة أخرى، يمكن القيام بذلك أثناء تنفيذ استعلام قاعدة بيانات واحد:

```
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:

بالإضافة إلى استرداد جميع السجلات المطابقة لاستعلام معين، يمكنك أيضًا استرداد سجلات فردية باستخدام الدوال find أو first أو firstWhere.
بدلاً من إرجاع مجموعة من النماذج، تقوم هذه الدوال بإرجاع مثيل نموذج واحد:

```
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:

في بعض الأحيان قد ترغب في تنفيذ بعض الإجراءات الأخرى إذا لم يتم العثور على نتائج.
ستعيد الدالتين findOr وfirstOr كائنا واحدًا للنموذج ، إذا لم يتم العثور على نتائج، فستنفذان الـ closure المحدد.
سيتم اعتبار القيمة التي يتم إرجاعها بواسطة الإغلاق نتيجة للدالة:

```
$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:

في بعض الأحيان قد ترغب في تنفيذ او رمي استثناء إذا لم يتم العثور على نموذج.
وهذا مفيد بشكل خاص في المسارات أو وحدات التحكم.
ستقوم الدالتين findOrFail وfirstOrFail باسترداد النتيجة الأولى للاستعلام؛ ومع ذلك، إذا لم يتم العثور على أي نتيجة، فسيتم رمي الـ Illuminate\Database\Eloquent\ModelNotFoundException:

```
$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:

إذا لم يتم اكتشاف ModelNotFoundException، فسيتم إرسال استجابة HTTP 404 تلقائيًا إلى العميل:

```
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:

الدالة **firstOrCreate**في Laravel تحاول البحث عن سجل في قاعدة البيانات باستخدام أزواج العمود / القيمة المعطاة.
إذا لم يتم العثور على النموذج في قاعدة البيانات، سيتم إدخال سجل جديد بخصائص ناتجة عن دمج الصفيف الأول (المعطيات التي يتم البحث بها) مع الصفيف الثاني الاختياري (المعطيات الإضافية المراد إدخالها إذا لم يتم العثور على السجل).

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:

الدالة **firstOrNew** في Laravel تشبه الدالة **firstOrCreate** حيث تحاول العثور على سجل في قاعدة البيانات باستخدام الخصائص المعطاة.
ومع ذلك، إذا لم يتم العثور على السجل، ستعيد **firstOrNew** كائنًا جديدًا للنموذج مع هذه الخصائص، لكنه **لن يتم حفظه في قاعدة البيانات تلقائيًا**.
سيتعين عليك استدعاء الدالة **save** يدويًا لحفظ السجل في قاعدة البيانات.

```
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:

عند التعامل مع نماذج **Eloquent**في Laravel ، يمكنك استخدام دوال التجميع مثل **count**، **sum**، **max**، وغيرها التي يوفرها **query builder**.كما هو متوقع، هذه الدوال تُرجع قيمة **scalar**(عدد أو مجموع أو قيمة قصوى، إلخ) بدلاً من كائن نموذج **Eloquent**.

```
$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:

بالطبع، عند استخدامEloquent ، لا يقتصر الأمر على استرجاع النماذج من قاعدة البيانات، بل نحتاج أيضًا إلى إدخال سجلات جديدة.

لحسن الحظ، يجعل **Eloquent** هذه العملية بسيطة جدًا.
لإدخال سجل جديد في قاعدة البيانات، يجب إنشاء كائن جديد من النموذج، وتعيين الخصائص على هذا النموذج، ثم استدعاء الدالة **save** على كائن النموذج لحفظ السجل في قاعدة البيانات.

```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
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.

في هذا المثال، نقوم بتعيين حقل **name** القادم من طلب HTTP إلى الخاصية **name** في كائن النموذج **App\Models\Flight**.
عند استدعاء الدالة **save**، سيتم إدخال سجل في قاعدة البيانات.
ستتم تلقائيًا إعداد حقول الطابع الزمني **created_at** و **updated_at** عند استدعاء الدالة **save**، لذلك لا حاجة لتعيين هذه الحقول يدويًا.

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:

بدلاً من استخدام الطريقة التقليدية بإنشاء كائن نموذج وتعيين الخصائص يدويًا، يمكنك استخدام الدالة **create** لحفظ نموذج جديد باستخدام سطر برمجي واحد فقط في PHP .
عند استخدام **create**، سيتم إرجاع كائن النموذج الذي تم إدخاله إلى قاعدة البيانات.

```
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.

قبل استخدام دالة **create** في Eloquent ، يجب عليك تحديد إما خاصية **$fillable** أو **$guarded** في فئة النموذج الخاصة بك.
هذه الخصائص مطلوبة لأن جميع نماذج Eloquent محمية افتراضيًا ضد هجمات (**mass assignment**) التعيين الجماعي غير الآمن

ما هو Mass Assignment؟
Mass Assignment يشير إلى القدرة على تعيين قيم متعددة لخصائص نموذج قاعدة البيانات دفعة واحدة بدون حماية.

قد يؤدي السماح بالتعيين الجماعي بدون قيود إلى تحديث حقول حساسة عن طريق الخطأ، مثل الحقول الخاصة بالصلاحيات أو الحقول الأمنية.

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:

يمكن استخدام الدالة **save** أيضًا لتحديث النماذج التي توجد بالفعل في قاعدة البيانات.
لتحديث نموذج موجود، يجب أولاً استرجاع النموذج من قاعدة البيانات، ثم تعيين أي خصائص ترغب في تعديلها.
بعد ذلك، يجب استدعاء طريقة **save** للنموذج. مرة أخرى، سيتم تحديث حقل **updated_at** تلقائيًا، لذلك لا حاجة لتعيينه يدويًا.

```
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.

أحيانًا قد تحتاج إلى تحديث نموذج موجود أو إنشاء نموذج جديد إذا لم يكن هناك نموذج مطابق.
مثل الدال **firstOrCreate**، تقوم الدالة **updateOrCreate** بحفظ النموذج تلقائيًا، لذلك لا حاجة لاستدعاء طريقة **save** يدويًا.

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:

في هذا المثال، إذا كانت هناك رحلة جوية (flight) موجودة مع موقع مغادرة **Oakland** ووجهة **San Diego**، فسيتم تحديث أعمدة **price** و **discounted** الخاصة بها.
وإذا لم تكن هذه الرحلة موجودة، فسيتم إنشاء رحلة جديدة باستخدام الخصائص الناتجة عن دمج الصفيف الأول مع الصفيف الثاني.

```
$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:

يمكن أيضًا إجراء التحديثات على النماذج التي تتطابق مع استعلام معين.
في هذا المثال، سنقوم بتحديث جميع الرحلات الجوية (flights) التي تكون نشطة (active) ولها وجهة **San Diego**، وسنقوم بتمييزها كمتأخرة(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.

تتوقع الدالة **update** في Eloquent مصفوفة مترابطة من أزواج العمود والقيمة، حيث تمثل الأعمدة التي يجب تحديثها.
ستقوم هذه الدالة بتحديث السجلات المتطابقة في قاعدة البيانات، وتعيد عدد الصفوف المتأثرة بالتحديث.

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.

عند تنفيذ تحديث جماعي عبر **Eloquent**، لن يتم إطلاق أحداث النماذج مثل **saving** و **saved** و **updating** و **updated** للنماذج التي تم تحديثها. يحدث ذلك لأن النماذج لا يتم استرجاعها فعليًا من قاعدة البيانات عند إجراء تحديث جماعي.

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.

تقدم Eloquent دوالا مثل **isDirty** و **isClean** و **wasChanged** لفحص الحالة الداخلية للنموذج وتحديد كيفية تغير خصائصه منذ أن تم استرجاع النموذج في الأصل.
هذه الدوال مفيدة للغاية لفهم ما إذا كانت الخصائص قد تغيرت قبل حفظ النموذج.

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".

تحدد الدالة **isDirty** ما إذا كانت أي من خصائص النموذج قد تغيرت منذ استرجاعه.
يمكنك تمرير اسم خاصية معينة أو مصفوفة من الخصائص إلى الدالة **isDirty** لتحديد ما إذا كانت أي من الخصائص "غير نظيفة(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:

تحدد الدالة **isClean** ما إذا كانت خاصية معينة قد بقيت دون تغيير منذ استرجاع النموذج. إذا لم يتم تعديل الخاصية، فإن هذه الدالة تعيد **true**، مما يعني أن الحالة "نظيفة". يمكنك أيضًا تمرير اسم خاصية معينة كمعامل اختياري إلى **isClean**للتحقق مما إذا كانت هذه الخاصية غير متغيرة.

```
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:

تحدد الدالة **wasChanged**ما إذا كانت أي من خصائص النموذج قد تغيرت عندما تم حفظ النموذج آخر مرة خلال دورة الطلب الحالية. هذه الدالة مفيدة لفهم ما إذا كانت هناك تغييرات تم حفظها على النموذج.

```
$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:

تُعيد الدالة **getOriginal** مصفوفة تحتوي على الخصائص الأصلية للنموذج بغض النظر عن أي تغييرات طرأت عليه منذ استرجاعه. إذا لزم الأمر، يمكنك تمرير اسم خاصية معينة إلى **getOriginal** للحصول على القيمة الأصلية لتلك الخاصية.

```
$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

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:

يمكنك استخدام الدالة **create** لحفظ نموذج جديد باستخدام سطر PHP واحد. ستقوم هذه الدالة بإدخال السجل في قاعدة البيانات وتعيد كائن النموذج الذي تم إدخاله.

```
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.

قبل استخدام الدالة **create** في Eloquent ، يجب عليك تحديد إما خاصية **$fillable** أو **$guarded** في فئة النموذج الخاصة بك. هذه الخصائص مطلوبة لأن جميع نماذج Eloquent محمية افتراضيًا ضد هجمات **mass assignment** التعيين الجماعي غير الآمن.

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.

تحدث ثغرة **mass assignment** عندما يقوم المستخدم بإرسال حقل طلب HTTP غير متوقع، مما يؤدي إلى تغيير عمود في قاعدة البيانات لم تكن تتوقعه. إذا لم يتم تطبيق الحماية بشكل صحيح، فقد يتمكن المستخدم الضار من تعديل الخصائص الحساسة للنموذج، مما يؤدي إلى نتائج غير مرغوب فيها.

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:

لتبدأ في حماية نموذجك من ثغرات **mass assignment**، يجب عليك تحديد الخصائص التي ترغب في جعلها قابلة للتعيين الجماعي. يمكنك القيام بذلك باستخدام خاصية **$fillable** في النموذج.

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{
   /**
    * The attributes that are mass assignable.
    *
    * @var array
    */
   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:

بمجرد تحديد الخصائص التي يمكن تعيينها بشكل جماعي باستخدام خاصية **$fillable**، يمكنك استخدام الدالة **create** لإدخال سجل جديد في قاعدة البيانات. ستقوم **create** بإرجاع كائن النموذج الذي تم إنشاؤه حديثًا.

```
$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:

إذا كان لديك كائن نموذج (model instance) بالفعل، يمكنك استخدام الدالة **fill** لتعبئته بمصفوفة من الخصائص. تعتبر الدالة **fill** مفيدة عندما ترغب في تحديث نموذج موجود بقيم جديدة دون الحاجة إلى إنشاء كائن جديد.

```
$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:

عند التعامل مع أعمدة JSON في Laravel ، يجب تحديد المفتاح القابل للتعيين الجماعي لكل عمود في مصفوفة **$fillable** في النموذج الخاص بك. يُعد هذا أمرًا ضروريًا لأن Laravel لا يدعم تحديث الخصائص المتداخلة (nested attributes) في JSON عند استخدام خاصية **$guarded** لأسباب أمنية.

```
/**
 * The attributes that are mass assignable.
 *
 * @var array
```

```
 */
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:

إذا كنت ترغب في جعل جميع خصائص النموذج قابلة للتعيين الجماعي، يمكنك تعريف خاصية **$guarded** في النموذج الخاص بك كمصفوفة فارغة. هذا يعني أنه سيتم السماح بتعيين جميع الخصائص بشكل جماعي، مما يمنحك مرونة أكبر عند إدخال البيانات.

```
/**
 * The attributes that aren't mass assignable.
 *
 * @var array
 */
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.

بشكل افتراضي، يتم تجاهل الخصائص التي لم تُدرج في مصفوفة **$fillable** بشكل صامت عند تنفيذ عمليات التعيين الجماعي(mass-assignment operations) .
هذا السلوك يُعتبر سلوكًا متوقعًا في بيئات الإنتاج، حيث يساهم في حماية النماذج من التعيين الجماعي غير الآمن.
ومع ذلك، قد يؤدي ذلك إلى حدوث ارتباك أثناء تطوير التطبيقات محليًا، حيث قد يتساءل المطورون لماذا لا تؤثر التغييرات على النموذج.

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:

إذا كنت ترغب في تعليم Laravel لرمي استثناء عند محاولة تعبئة خاصية غير قابلة للتعيين(unfillable attribute) ، يمكنك استدعاء الدالة **preventSilentlyDiscardingAttributes**. عادةً ما يتم استدعاء هذه الدالة في دالة **boot** في فئة **AppServiceProvider** الخاصة بتطبيقك.

```
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.

تُستخدم دالة **upsert** في Eloquent لإجراء عملية تحديث أو إدراج للسجلات في عملية واحدة متكاملة(atomic operation) .
تساعد هذه الطريقة في تحسين الأداء عند الحاجة إلى تحديث أو إنشاء سجلات متعددة في قاعدة البيانات.

كيفية عمل :upsert

- **المعلمة الأولى** :تحتوي على القيم التي سيتم إدخالها أو تحديثها.
- **المعلمة الثانية** :تحدد الأعمدة التي تميز السجلات بشكل فريد في الجدول (مثل الأعمدة الرئيسية أو الأعمدة التي تمثل معرفات فريدة).
- **المعلمة الثالثة (اختيارية)** :تحدد الأعمدة التي سيتم تحديثها إذا تم العثور على سجل مطابق.

The upsert method will automatically set the created_at and updated_at timestamps if timestamps are enabled on the model:

عند استخدام الدالة **upsert** في Eloquent ، سيتم تلقائيًا تعيين قيم الطابع الزمني **created_at**و **updated_at**إذا كانت الطوابع الزمنية مفعلة على النموذج. هذا يعني أن **upsert**يتعامل مع تحديث وإنشاء السجلات بطريقة تجعل هذه الحقول مُدارة تلقائيًا من قبلLaravel .

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

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.

عند استخدام الدالة **upsert**فيEloquent ، هناك متطلبات خاصة تتعلق بفهرسة الأعمدة في قاعدة البيانات:

1. **جميع قواعد البيانات باستثناءSQL Server** :
   - تتطلب أن تكون الأعمدة المحددة في المعلمة الثانية من **upsert**(الأعمدة التي تحدد السجلات الفريدة) تحتوي على فهرس **primary**أو **unique**.
   - هذا يعني أنك تحتاج إلى التأكد من أن الأعمدة المحددة في المعلمة الثانية (مثل **name**أو **email**) تحتوي على فهرس يضمن أن تكون هذه الأعمدة فريدة في الجدول.
2. **MariaDB و MySQL** :
   - كلاهما يتجاهل المعلمة الثانية من **upsert**(الأعمدة التي تحدد السجلات الفريدة) ويستخدم دائمًا الفهارس **primary**و **unique** الموجودة في الجدول لتحديد السجلات المطابقة.
   - لذلك، حتى لو حددت أعمدة في المعلمة الثانية، سيعتمد MySQL و MariaDB على الفهارس المحددة في الجدول فقط.

ماذا يعني هذا عمليًا؟

1. **لجميع قواعد البيانات باستثناءSQL Server** :
   - يجب أن تتأكد من أن الأعمدة التي تحددها في المعلمة الثانية (مثل **email**أو **name**) تم تعريفها بفهرس **primary** أو **unique.**

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();
```

You may call the truncate method to delete all of the model's associated database records. The truncate operation will also reset any auto-incrementing IDs on the model's associated table:

يمكنك استدعاء الدالة **truncate** لحذف جميع السجلات المرتبطة بجدول النموذج في قاعدة البيانات.

بالإضافة إلى حذف جميع السجلات، ستقوم عملية **truncate** أيضًا بإعادة تعيين أي معرفات تعتمد على التسلسل التلقائي (**auto-incrementing IDs**) في الجدول المرتبط بالنموذج(مثل العمود **id** لتبدأ من الترقيم من البداية أو القيمة الافتراضية (1) عند إدراج سجلات جديدة في الجدول).

```
Flight::truncate();
```

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:

في المثال السابق، قمنا باسترجاع النموذج من قاعدة البيانات قبل استدعاء الدالة **delete.** ولكن إذا كنت تعرف المفتاح الأساسي (primary key) للنموذج، يمكنك حذف النموذج دون استرجاعه صراحةً باستخدام الدالة **destroy.**

الدالة **destroy** تتيح لك حذف سجل أو أكثر باستخدام المفتاح الأساسي فقط (لأنها تستقبل مفتاح رئيسي واحد او مصفوفة او مجموعة من المفاتيح الرئيسية)، دون الحاجة إلى تحميل النموذج من قاعدة البيانات.

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

ما هو Soft Deleting؟

Soft Deleting هي ميزة في Laravel تتيح لك "حذف" السجلات دون إزالتها فعليًا من قاعدة البيانات.
بدلاً من حذف السجل، يتم تعيين حقل **deleted_at** إلى الوقت الحالي (الذي تم فيه حذف السجل)، مما يجعل السجل "غير مرئي" في الاستعلامات العادية، ولكنه لا يزال موجودًا في قاعدة البيانات.

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

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

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.

الدالة **destroy** في Laravel تقوم بتحميل كل نموذج (model) بشكل فردي ثم تستدعي الدالة **delete** على كل نموذج على حدة، مما يضمن إطلاق أحداث **deleting** و **deleted** لكل نموذج يتم حذفه.

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:

بالطبع، يمكنك بناء استعلام **Eloquent** لحذف جميع النماذج التي تتطابق مع معايير الاستعلام الخاص بك.
عندما تقوم بحذف عدة سجلات دفعة واحدة باستخدام **mass delete**، لن يتم إطلاق أحداث النماذج (**model events**) مثل **deleting** و **deleted**، كما يحدث في حالة **mass update**.

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

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.

عند تنفيذ عملية **mass delete** باستخدام Eloquent ، لن يتم إطلاق أحداث النموذج **deleting** و **deleted** للسجلات المحذوفة.
والسبب في ذلك هو أن السجلات لا يتم استرجاعها فعليًا من قاعدة البيانات قبل حذفها عند استخدام استعلام الحذف الجماعي.
شرح السبب:

- **mass delete**يعمل مباشرة على مستوى قاعدة البيانات باستخدام استعلامSQL ، دون الحاجة إلى تحميل السجلات ككائنات **Eloquent models**.
- نظرًا لأن السجلات لا يتم تحميلها ككائنات، لا يمكن لـ Eloquent إطلاق أحداث **deleting**و **deleted**، لأن هذه الأحداث مرتبطة بتحميل وتفاعل النماذج مع التطبيق.

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:

في Eloquent ، بالإضافة إلى حذف السجلات بشكل نهائي من قاعدة البيانات، يمكنك أيضًا تفعيل **Soft Deletes** للنماذج. عندما يتم حذف نموذج بهذا الاسلوب، فإنه لا يتم حذفه فعليًا من قاعدة البيانات. بدلاً من ذلك، يتم إعطاء قيمة لعمود **deleted_at** تشير إلى تاريخ ووقت الحذف.

```
<?php
namespace App\Models;
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.

عند استخدام **SoftDeletes** في Eloquent ، سيتم تلقائيًا تحويل (cast) الخاصية **deleted_at**إلى كائن **DateTime**أو **Carbon**عند التعامل معها. هذا يجعل من السهل التحقق من الأوقات المرتبطة بالحذف المؤقت باستخدام وظائف **Carbon**، والتي توفر الكثير من الميزات للتعامل مع التواريخ والأوقات.

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

إضافة عمود **deleted_at**إلى جدول قاعدة البيانات عند استخدام **Soft Deletes**في Laravel ، يتم باستخدام **Laravel Schema Builder** الذي يحتوي على دالة مساعدة لإنشاء هذا العمود بسهولة. هذه الدالة هي **softDeletes()**، والتي تضيف عمود **deleted_at**إلى الجدول تلقائيًا.

```
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.

عند استدعاء الدالة **delete**على نموذج يستخدم **Soft Deletes**في Laravel ، سيتم تحديد قيمة للعمود **deleted_at** إلى التاريخ والوقت الحاليين، بدلاً من حذف السجل فعليًا من قاعدة البيانات. ومع ذلك، يبقى السجل في الجدول ويُعتبر "محذوفًا" لأنه يحتوي على قيمة في **deleted_at**.

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:

في بعض الأحيان، قد ترغب في "استعادة" نموذج محذوف باستخدام **Soft Deletes**، أي استعادة السجل المحذوف مؤقتًا إلى الحالة النشطة. يمكنك استخدام الدالة **restore**على النموذج لاستعادة هذا السجل. عند استدعاء **restore**، سيتم تعيين قيمة العمود **deleted_at** إلى **null**، مما يعني أن السجل لم يعد محذوفًا وسيتم استرجاعه في استعلامات Eloquent العادية.

```
$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:

تستطيع أيضا استخدام الدالة restore لاسترجاع أكثر من سجل محذوف. مثل كل عمليات الـ mass الاسترجاع الجماعي لا يطلق أي جدث للنموذج الذي نسترجعه.

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

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

نستطيع استخدام الدالة restore في استعلامات استرجاع السجلات المحذوفة من جدول له علاقة بسجل او سجلات في جدول اخر.

```
$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:

أحيانا قد ترغب في حذف النموذج (السجل) بشكل دائم من قاعدة البيانات. يمكنك استخدام الدالة forceDelete للحذف الدائم من قاعدة البيانات.

```
$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:

في الامثلة السابقة لاحظنا ان نماذج السجلات المحذوفة باستخدام الـ soft delete يتم استثناؤها وعدم استرجاعها في نتيجة استعلام الاسترجاع. ولكي يتم استرجاع هذه السجلات تستطيع استخدام الدالة withTrashed.

```
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:

الدالة onlyTrashed تسترجع فقط النماذج المحذوفة باستخدام الـ soft delete

```
$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.

في بعض الأحيان قد ترغب في حذف النماذج التي لم تعد بحاجة إليها بشكل دوري. لتحقيق ذلك، يمكنك استخدام **trait**في Laravel الذي يساعدك على تنظيف السجلات القديمة أو غير الضرورية Laravel .يوفر **Prunable**و **MassPrunable**لتحقيق هذا الهدف.

الخيارات المتاحة:

1. **Illuminate\Database\Eloquent\Prunable:** يقوم هذا **trait**بحذف النماذج التي لم تعد ضرورية بشكل فردي. يتم استدعاء الأحداث المرتبطة بالنموذج مثل **deleting**و .**deleted**
2. **Illuminate\Database\Eloquent\MassPrunable:**هذا **trait**يقوم بحذف السجلات على دفعات دون استرجاع النماذج ككائنات Eloquent، مما يجعله أكثر كفاءة لحذف كميات كبيرة من السجلات **MassPrunable**. لا يستدعي أحداث الحذف مثل **deleting**و .**deleted**

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:

بعد إضافة أحد Prunable أو MassPrunable إلى النموذج، يجب عليك تنفيذ دالة **prunable**، والتي تُرجع كائن **Eloquent query builder**. هذا الاستعلام يقوم بتحديد السجلات التي لم تعد ضرورية بناءً على الشروط التي تحددها.

```
<?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:

عند تحديد النماذج على أنها **Prunable**فيLaravel ، يمكنك أيضًا تعريف دالة **pruning**على النموذج. يتم استدعاء هذه الدالة قبل حذف النموذج فعليًا، وتُعتبر مفيدة إذا كنت بحاجة إلى حذف موارد إضافية مرتبطة بالنموذج (مثل الملفات المخزنة، أو السجلات المرتبطة) قبل أن يتم حذف السجل من قاعدة البيانات بشكل دائم.

```
/**
 * 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:

بعد تكوين نموذج **Prunable**، يمكنك جدولة أمر **Artisan**الخاص بـ **model** لتنفيذ عملية الحذف الدورية للنماذج التي لم تعد ضرورية. يتم تنفيذ هذا الأمر بشكل دوري باستخدام **scheduler**في.Laravel

```
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:

أمر **model:prune** في Laravel يقوم تلقائيًا بالكشف عن النماذج التي تستخدم **Prunable**داخل مجلد **app/Models**لتطبيق عملية التنظيف (pruning).ولكن إذا كانت النماذج الخاصة بك موجودة في مسار مختلف أو في مجلد آخر داخل التطبيق، يمكنك استخدام الخيار **model--**لتحديد أسماء فئات النماذج التي ترغب في تشغيل عملية **prune**عليها.

```
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:

إذا كنت ترغب في استثناء نماذج معينة من عملية **pruning**أثناء تنظيف جميع النماذج الأخرى المكتشفة تلقائيًا بواسطة أمر **model:prune**، يمكنك استخدام الخيار .**except--**يتيح لك هذا الخيار استثناء نماذج محددة من عملية الحذف، مع السماح لبقية النماذج المكتشفة بتطبيق عملية التنظيف عليها.

```
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:

يمكنك اختبار استعلام **prunable** الخاص بك باستخدام خيار **--pretend** عند تنفيذ أمر **model:prune**. عند استخدام هذا الخيار، لن يتم حذف أي سجلات بالفعل، بل سيقوم الأمر بالإبلاغ عن عدد السجلات التي سيتم حذفها لو تم تشغيل الأمر فعليًا.

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

Soft deleting models will be permanently deleted (forceDelete) if they match the prunable query.

عند استخدام **soft deleting** مع **Prunable** في Laravel ، سيتم **حذف النماذج بشكل دائم** (استخدام **forceDelete**) إذا كانت هذه النماذج تتطابق مع استعلام **prunable** الخاص بك. بعبارة أخرى، حتى النماذج التي تم حذفها مؤقتًا باستخدام **soft delete** سيتم إزالتها بشكل دائم من قاعدة البيانات إذا تم تشغيل عملية **prune** عليها.

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:

عند استخدام **trait** الخاص بـ **Illuminate\Database\Eloquent\MassPrunable** مع النماذج في Laravel ، يتم حذف السجلات باستخدام استعلامات حذف جماعية (mass-deletion queries) مباشرة من قاعدة البيانات. وهذا يجعل عملية **pruning** أكثر كفاءة لأن السجلات لا يتم استرجاعها كنماذج **Eloquent** قبل حذفها. نتيجة لذلك:

1. **دالة pruning**: لن يتم استدعاؤها.
2. **أحداث deleting** و**deleted**: لن يتم إطلاقها.

```
<?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:

في Laravel ، يمكنك إنشاء نسخة غير محفوظة من كائن نموذج موجود باستخدام طريقة **replicate**. هذه الطريقة مفيدة بشكل خاص عندما يكون لديك كائنات نموذج تشترك في العديد من الخصائص، وتريد إنشاء نسخة مماثلة مع تغييرات طفيفة على بعض الحقول قبل حفظها.

كيفية استخدام replicate:

- **replicate** تنشئ نسخة من النموذج الحالي بدون حفظه في قاعدة البيانات. يمكنك بعد ذلك تعديل الخصائص إذا لزم الأمر وحفظ النسخة الجديدة.

```
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:

لإستبعاد خاصية أو أكثر من عملية النسخ عند استخدام الدالة **replicate** في Laravel ، يمكنك تمرير مصفوفة تحتوي على أسماء الخصائص التي لا تريد نسخها إلى النموذج الجديد. يتم استبعاد هذه الخصائص من النسخة الجديدة، ويمكنك بعد ذلك حفظ النموذج بدون تلك القيم.

```
$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));
    }
}
```

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:

```php
<?php
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:

```
<?php
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
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:

```
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:

```
$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:

```
$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();
```

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

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:

```
<?php
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.

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:

```
<?php
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
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);
}
```

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
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:

```
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();
```