فراس الدبعي lecture 06 Database الملزمة الأولى وشرح الاكواد الغير واضحة

Database:

(مقدمة) Introduction

يُسهل إطار العملLaravel عملية التفاعل مع قواعد البيانات في تطبيقات الويب الحديثة، من خلال تقديم تكامل سلس مع العديد من أنظمة ادارة قواعد البيانات المدعومة. ويوفر عدة طرق للعمل مع قواعد البيانات، مثل:

- **استعلامات SQL الخام** :تنفيذ مباشر لأوامرSQL .
- **منشئ الاستعلامات المرن** :واجهة مرنة يمكن تسلسلها لبناء استعلامات SQL برمجيًا.
- **Eloquent ORM**: أداة ربط الكائنات بالعلاقات (ORM) التي تتيح للمطورين التفاعل مع سجلات قواعد البيانات باستخدام كائناتPHP .

حاليًا، يقدم Laravel دعمًا رسميًا للعديد من قواعد البيانات:

١. **MariaDB**الإصدار ١٠,٣+
٢. **MySQL**الإصدار ٥,٧+
٣. **PostgreSQL**الإصدار ١٠,٠+
٤. **SQLite** الإصدار ٣,٢٦,٠+
٥. **SQL Server**الإصدار ٢٠١٧+

توفر هذه المرونة العديد من الخيارات لإطارLaravel ، مما يجعله خيارًا قويًا للتطبيقات التي تتطلب إدارة قواعد البيانات عبر منصات مختلفة.

(التهيئة) Configuration

يتم تخزين إعدادات خدمات قاعدة البيانات الخاصة بـ Laravel في الملف **config/database.php** الخاص بالتطبيق الذي تطوره. في هذا الملف، يمكنك تعريف جميع اتصالات قواعد البيانات الخاصة بك، بالإضافة إلى تحديد الاتصال الافتراضي الذي يجب استخدامه. تعتمد معظم خيارات التكوين في هذا الملف على قيم متغيرات البيئة الخاصة بالتطبيق.

يتم توفير أمثلة لمعظم أنظمة قواعد البيانات المدعومة من Laravel داخل هذا الملف. بشكل افتراضي، يكون تكوين بيئة Laravel الجاهزة معدًا للاستخدام مع **Laravel Sail**، وهي إعداد Docker مخصص لتطوير تطبيقات Laravel على جهازك المحلي. ومع ذلك، يمكنك تعديل إعدادات قاعدة البيانات الخاصة بك حسب الحاجة لتتناسب مع قاعدة البيانات المحلية الخاصة بك.

تكوين SQLite (SQLite Configuration)

قواعد بيانات **SQLite**تُحفظ في ملف واحد داخل نظام الملفات الخاص بك. يمكنك إنشاء قاعدة بيانات جديدة باستخدام أمر **touch**في الطرفية الخاصة بك كما يلي:

```
touch database/database.sqlite // إنشاء قاعدة بيانات SQLite جديدة
```

بعد إنشاء قاعدة البيانات، يمكنك بسهولة تكوين متغيرات البيئة الخاصة بك للإشارة إلى هذه القاعدة من خلال وضع المسار المطلق للقاعدة في متغير البيئة **DB_DATABASE**كما يلي:

في ملف **env.**الخاص بتطبيقك، قم بتحديث السطر التالي ليشير إلى مسار قاعدة البيانات:

```
DB_CONNECTION=sqlite // نوع الاتصال بقاعدة البيانات
DB_DATABASE=/absolute/path/to/database.sqlite
```

استبدل **absolute/path/to/your/database.sqlite**/بالمسار الكامل للملف الذي قمت بإنشائه باستخدام أمر .**touch**

بشكل افتراضي، يتم تمكين قيود المفاتيح الاجنبية (foreign key constraints) لاتصالات **SQLite**. إذا كنت ترغب في تعطيلها، يجب عليك تعيين متغير البيئة **DB_FOREIGN_KEYS** إلى **false** في ملف **env.**الخاص بك كما يلي:

```
DB_FOREIGN_KEYS=false
```

سيؤدي هذا إلى تعطيل قيود المفاتيح الاجنبية لقاعدة البيانات SQLite في تطبيقLaravel .

(تكوينMicrosoft SQL Server) Microsoft SQL Server Configuration

لاستخدام قاعدة بيانات **Microsoft SQL Server** معLaravel ، يجب عليك التأكد من تثبيت امتدادات PHP التالية:

- **sqlsrv**
- **pdo_sqlsrv**

بالإضافة إلى ذلك، ستحتاج إلى تثبيت أي تبعيات ضرورية لهذه الامتدادات، مثل .**Microsoft SQL ODBC driver**

يمكنك التحقق من تثبيت الامتدادات باستخدام الأمر التالي في الطرفية:

```
php -m | grep sqlsrv
php -m | grep pdo_sqlsrv
```

إذا لم تكن هذه الامتدادات مثبتة، يمكنك تثبيتها باستخدام مدير الحزم المناسب لنظام التشغيل الخاص بك.

التكوين باستخدام الروابط(Configuration Using URLs)

عادةً ما يتم تكوين اتصالات قواعد البيانات باستخدام عدة قيم تكوينية مثل **host** المضيف، **database** قاعدة البيانات، **username** اسم المستخدم، **password** كلمة المرور، وما إلى ذلك. كل من هذه القيم التكوينية لها متغير بيئة خاص بها. وهذا يعني أنه عند تكوين معلومات اتصال قاعدة البيانات على خادم الإنتاج، ستحتاج إلى إدارة عدة متغيرات بيئية.

ومع ذلك، بعض مقدمي خدمات قواعد البيانات المُدارة مثل **AWS** و **Heroku** يوفرون **"URL"لقاعدة البيانات** يحتوي على جميع معلومات الاتصال في سلسلة نصية واحدة. قد يبدو الـ **URL** لقاعدة البيانات كما يلي:

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

في هذا المثال:

- **mysql**: محرك او نظام ادارة قاعدة البيانات
- **Username**: اسم المستخدم لاتصال قاعدة البيانات
- **password**: كلمة المرور
- **host**: المضيف غالبًا عنوان IP أو اسم النطاق
- **Port**: المنفذ الذي يستمع إليه الخادم
- **Database**: اسم قاعدة البيانات

يمكنك تعيين هذا **URL** في متغير البيئة الخاص بـ Laravel واستخدامه لتبسيط تكوين الاتصال بقاعدة البيانات.

للتسهيل، يدعم Laravel استخدام **URLs** كبديل لتكوين قاعدة البيانات باستخدام خيارات تكوين متعددة. إذا كان خيار التكوين **url** أو متغير البيئة المقابل **DB_URL** موجودًا، فسيتم استخدامه لاستخراج معلومات الاتصال بقاعدة البيانات وبيانات الاعتماد.

على سبيل المثال، بدلاً من إعداد متغيرات بيئة منفصلة لكل من **DB_HOST**، و**DB_DATABASE**، و**DB_USERNAME**، و**DB_PASSWORD**، يمكنك ببساطة استخدام متغير **DB_URL**في ملف **env.**الخاص بك كما يلي:

```
DB_URL=mysql://username:password@host:port/database
```

سيقوم Laravel تلقائيًا بتحليل هذا الـ **URL**لاستخراج التفاصيل اللازمة لاتصال قاعدة البيانات، مما يجعل التكوين أكثر بساطة خاصةً عند استخدام مزودي خدمات قواعد البيانات المُدارة مثل **AWS** و **Heroku**.

<u>اتصالات القراءة والكتابة(Read and Write Connections)</u>

أحيانًا قد ترغب في استخدام اتصال قاعدة بيانات واحد لتنفيذ استعلامات **SELECT**، واستخدام اتصال آخر لتنفيذ استعلامات **INSERT** و **UPDATE** و **DELETE** . يوفر Laravel هذه الميزة بسهولة، وسيتم استخدام الاتصالات المناسبة دائمًا سواء كنت تستخدم الاستعلامات الخام(raw queries) ، أو منشئ الاستعلامات(query builder) ، أو.Eloquent ORM

```
'mysql' => [
    'read' => [ // اتصالات القراءة
        'host' => [
            '192.168.1.1', // خادم القراءة الأول
            '196.168.1.2', // خادم القراءة الثاني
        ],
    ],
    'write' => [ // اتصالات الكتابة
        'host' => [
            '196.168.1.3', // خادم الكتابة
        ],
    ],
    'sticky' => true, // تمكين الخيار اللاصق
    'database' => env('DB_DATABASE', 'laravel'), // اسم قاعدة البيانات
    'username' => env('DB_USERNAME', 'root'),    // اسم المستخدم
    'password' => env('DB_PASSWORD', ''),        // كلمة المرور
    'unix_socket' => env('DB_SOCKET', ''),       // مقبس يونكس
    'charset' => env('DB_CHARSET', 'utf8mb4'),   // مجموعة الأحرف
    'collation' => env('DB_COLLATION', 'utf8mb4_unicode_ci'), // الترتيب
    'prefix' => '', // بادئة الجداول
    'prefix_indexes' => true, // بادئة الفهارس
    'strict' => true, // الوضع الصارم
    'engine' => null, // محرك التخزين
    'options' => extension_loaded('pdo_mysql') ? array_filter([
        PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'), // خيارات SSL
    ]) : [],
],
```

لاحظ أنه تم إضافة ثلاثة مفاتيح إلى مصفوفة التكوين:**read** ، **write**، و **sticky**. تحتوي مفاتيح **read** و **write**على قيم مصفوفات تحتوي على مفتاح واحد فقط هو **host**. بينما سيتم دمج بقية خيارات قاعدة البيانات لاتصالات القراءة والكتابة من مصفوفة التكوين الرئيسية الخاصة بـ **mysql**.

لا تحتاج إلى وضع العناصر في مصفوفات **read**و **write** إلا إذا كنت ترغب في تجاوز القيم الموجودة في المصفوفة الرئيسية الخاصة بـ **mysql**. لذا، في هذه الحالة، سيتم استخدام **192.168.1.1** كمضيف لاتصال **"read"**، بينما سيتم استخدام **192.168.1.3**لاتصال **"write"**.

ستتم مشاركة بيانات اعتماد قاعدة البيانات، والبادئة(prefix) ، ومجموعة الأحرف(character set) ، وجميع الخيارات الأخرى الموجودة في المصفوفة الرئيسية الخاصة بـ **mysql**عبر كلا الاتصالين.

كيف يعمل ذلك:

- **قراءة البيانات** :عند تنفيذ استعلام **SELECT**، سيستخدم Laravel المضيف .**192.168.1.1**
- **كتابة البيانات** :عند تنفيذ استعلامات **INSERT**، **UPDATE**، أو **DELETE**، سيستخدم المضيف .**192.168.1.3**
- **مشاركة الخيارات** :جميع الخيارات الأخرى مثل اسم قاعدة البيانات، واسم المستخدم، وكلمة المرور، ستبقى كما هي لكلا الاتصالين.

اختيار عشوائي للمضيف:

عندما تحتوي مصفوفة تكوين المضيف على قيم متعددة، سيتم اختيار مضيف قاعدة بيانات بشكل عشوائي لكل طلب. هذا يعني أنه إذا كان لديك أكثر من مضيف في مصفوفات **read**أو **write**، سيساهم ذلك في تحسين التحميل والتوازن عبر الخوادم المتاحة.

مثال توضيحي:

```
'mysql' => [
    'driver' => 'mysql',
    'database' => 'database_name',
    'username' => 'your_username',
    'password' => 'your_password',
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
    'strict' => true,
    'engine' => null,
    'read' => [ // اتصال القراءة
        'host' => ['192.168.1.1'],
    ],
    'write' => [ // اتصال الكتابة
        'host' => ['192.168.1.3'],
    ],
],
```

بهذه الطريقة، يمكنك ضبط اتصالات قاعدة البيانات في Laravel لتلبية احتياجات تطبيقك بشكل فعال، مما يتيح لك الاستفادة من الخوادم المتعددة لتحسين الأداء والموثوقية.

خيار اللاصق (The sticky Option)

خيار **sticky** هو قيمة اختيارية يمكن استخدامها للسماح بالقراءة الفورية للسجلات التي تمت كتابتها في قاعدة البيانات خلال دورة الطلب الحالية.

إذا تم تمكين خيار **sticky**وتم تنفيذ عملية **"write"** على قاعدة البيانات خلال دورة الطلب الحالية، فإن أي عمليات **"read"**لاحقة ستستخدم اتصال الـ **"write"**.

كيف يعمل ذلك:

- عند إجراء عملية كتابة (مثل **INSERT**أو **UPDATE**)، إذا كان خيار **sticky**مفعلًا، سيتم توجيه جميع عمليات القراءة اللاحقة في نفس دورة الطلب إلى نفس الاتصال الذي تم استخدامه للكتابة.
- هذا يضمن أنه يمكن قراءة أي بيانات تم كتابتها خلال دورة الطلب الحالية مباشرةً من قاعدة البيانات أثناء نفس الطلب.

تشغيل استعلامات SQL (Running SQL Queries)

بمجرد تكوين اتصال قاعدة البيانات الخاص بك، يمكنك تشغيل الاستعلامات باستخدام **DB facade**. يوفر **DB facade** دوالا لكل نوع من أنواع الاستعلامات : **select**، و **update**، و **insert**، و **delete**، و **statement**.

تشغيل استعلام SELECT (Running a Select Query)

المعامل الأول الذي تم تمريره إلى دالة **select**هي استعلامSQL ، بينما المعامل الثاني هي أي روابط معاملات تحتاج إلى الربط مع الاستعلام. عادةً، تكون هذه القيم هي قيم قيود جملة **where**. يوفر ربط المعاملات حماية ضد هجمات **SQL injection**.

مثال توضيحي:

```
$users = DB::select('SELECT * FROM users WHERE active = ?', [1]);
```

كيف يعمل:

- **استعلام** '**SQL**: 'SELECT * FROM users WHERE active = ?' هو استعلام SQL الذي يبحث عن المستخدمين النشطين.
- **روابط المعاملات** : [1]هي القيمة التي سيتم ربطها بالاستعلام. في هذه الحالة، يتم استبدال علامة الاستفهام **?**بالقيمة **1**، مما يعني أن الاستعلام سيبحث عن المستخدمين الذين يكون حقل **active** لديهم مساويًا لـ ١.

ستقوم الدالة **select**دائمًا بإرجاع مصفوفة من النتائج. كل نتيجة ضمن المصفوفة ستكون كائنًا من نوع **stdClass** في PHP تمثل سجلًا من قاعدة البيانات.

كيفية استخدام النتائج:

عند استخدام الدالة **select**، يمكنك التفاعل مع السجلات المستردة بسهولة. إليك مثالًا يوضح كيفية استخدام النتائج:

```
use Illuminate\Support\Facades\DB;
$users = DB::select('SELECT * FROM users WHERE active = ?', [1]);
foreach ($users as $user) {
   echo $user->name;  // طباعة اسم المستخدم
   echo $user->email; // طباعة بريد المستخدم الإلكتروني
}
```

كيف يعمل هذا:

- **إرجاع المصفوفة** :عند تنفيذ الاستعلام، يتم إرجاع مصفوفة تحتوي على كل السجلات التي تطابق الشروط المحددة.
- **كائنات** :**stdClass** كل عنصر في المصفوفة هو كائن يمثل سجلًا واحدًا. يمكنك الوصول إلى خصائص السجل باستخدام أسماء الأعمدة، مثل **$user->name**و. **$user->email**

ملاحظة:

- إذا لم يتم العثور على أي سجلات، ستعيد الدالة **select** مصفوفة فارغة.

بهذا الشكل، يوفر لك استخدام الدالة **select** في Laravel طريقة بسيطة وآمنة للتفاعل مع البيانات المستردة من قاعدة البيانات.

اختيار القيم المفردة(Selecting Scalar Values)

أحيانًا قد تسترجع استعلامات قاعدة البيانات قيمة مفردة (scalar value) بدلاً من مجموعة من السجلات. بدلاً من الحاجة إلى استرداد القيمة المفردة من كائن سجل، يتيح لك Laravel استرداد هذه القيمة مباشرة باستخدام الدالة **scalar**.

كيفية استخدام الدالة scalar:

يمكنك استخدام الدالة **DB::select** مع جملة SQL التي ترجع قيمة مفردة، مثل استخدام **COUNT**أو **SUM**.

مثال على استخدام الدالة scalar:

إذا كنت بحاجة إلى استرداد قيمة مفردة من قاعدة البيانات، يمكنك القيام بذلك كما يلي:

```
$activeUserCount = DB::select('SELECT COUNT(*) FROM users WHERE active = ?', [1])// استرجاع القيمة المفردة
$count = $activeUserCount[0]->count;
```

ملاحظات:

- في حالة استخدام الاستعلام المباشر مع **scalar**، يجب أن تتأكد من أن الاستعلام الذي تستخدمه يُرجع فقط قيمة مفردة، مثل استخدام دالة تجميع (aggregate function)مثل **SUM**، **AVG**، **COUNT**، إلخ.
- دالة **scalar**توفر وسيلة سريعة وسهلة لاسترداد القيم الفردية دون الحاجة للتعامل مع كائنات السجل.

باستخدام هذه الدوال، يمكنك استرداد القيم الفردية من قاعدة البيانات بسهولة وسرعة باستخدام.Laravel

اختيار مجموعات نتائج متعددة(Selecting Multiple Result Sets)

إذا كان تطبيقك يستدعي الإجراءات المخزنة (stored procedures) التي ترجع مجموعات نتائج متعددة، يمكنك استخدام الدالة **selectResultSets**لاسترجاع كل مجموعات النتائج التي يتم إرجاعها بواسطة الإجراء المخزن.

كيفية استخدام selectResultSets:

يمكنك استدعاء الإجراء المخزن الخاص بك باستخدام **selectResultSets**، وسيتم إرجاع جميع مجموعات النتائج في مصفوفة. إليك مثال يوضح كيفية استخدام هذه الطريقة:

```
$resultSets = DB::selectResultSets('CALL your_stored_procedure_name()');
```

كيفية التعامل مع النتائج:

بعد استدعاء **selectResultSets**، يمكنك التفاعل مع كل مجموعة نتائج بشكل منفصل. على سبيل المثال:

```
$resultSets = DB::selectResultSets('CALL your_stored_procedure_name()');
foreach ($resultSets as $resultSet) {
   foreach ($resultSet as $row)  // الوصول إلى البيانات في الصف
      echo $row->column_name; // استبدل column_name باسم العمود المناسب
   }
}
```

ملاحظات:

- تأكد من أن الإجراء المخزن الخاص بك مصمم لإرجاع مجموعات نتائج متعددة.
- كل مجموعة نتائج ستكون في شكل مصفوفة من كائنات **stdClass**، مما يسهل الوصول إلى البيانات.

مزايا استخدام selectResultSets :

١. **سهولة الوصول** :يمكنك بسهولة استرداد ومعالجة مجموعات النتائج المتعددة التي تعيدها الإجراءات المخزنة.
٢. **تحكم أفضل** :يمنحك القدرة على التعامل مع البيانات المسترجعة بشكل منظم ومنهجي.

باستخدام **selectResultSets**، يمكنك تحسين كيفية تعامل تطبيقك مع الإجراءات المخزنة ومجموعات النتائج المتعددة، مما يزيد من مرونة وكفاءة استعلاماتك فيLaravel .

استخدام الروابط المسماة (Using Named Bindings)

بدلاً من استخدام ? لتمثيل روابط المعاملات الخاصة بك، يمكنك تنفيذ استعلام باستخدام روابط مسماة (named bindings) فيLaravel . تعتبر الروابط المسماة أكثر وضوحًا وسهولة في القراءة، خاصةً عندما يكون لديك العديد من المعاملات.

كيفية استخدام الروابط المسماة:

عند استخدام الروابط المسماة، يمكنك تعيين أسماء للمعلمات في استعلام SQL واستخدام مصفوفة لتمرير القيم. إليك مثال يوضح ذلك:

```
$results = DB::select('select * from users where id = :id AND age > :age', ['id' => 1,  'age' => 18,]);
```

كيفية عمل ذلك:

- **في DB::select**، يتم استخدام AND كروابط مسماة للمعلمات.
- **مصفوفة المعاملات** :يتم تمرير القيم إلى استعلام SQL من خلال مصفوفة تحتوي على أسماء المعاملات وقيمها.

مزايا استخدام الروابط المسماة:

١. **الوضوح** :تجعل الروابط المسماة الاستعلامات أكثر وضوحًا وسهولة في الفهم.
٢. **سهولة الصيانة** :تسهل إضافة أو تعديل المعلمات، حيث يمكنك ببساطة تحديث القيمة المرتبطة باسم المعلمة.

مثال كامل:

```
$sql = 'SELECT * FROM users WHERE active = :active AND age > :age';
$parameters = [
   'active' => 1,
   'age' => 18,
];
$users = DB::select($sql, $parameters);
foreach ($users as $user) {
   echo $user->name;
}
```

باستخدام الروابط المسماة، يمكنك كتابة استعلامات SQL بطريقة أكثر وضوحًا وأمانًا، مما يسهل قراءة الكود وفهمه.

تشغيل جملة INSERT (Running an Insert Statement)

لتنفيذ جملة **INSERT**، يمكنك استخدام الدالة **insert**على **DB facade**. مثل دالة **select**، تقبل هذه الدالة استعلام SQL كمعامل أول وروابط كمعامل ثانٍ.

كيفية استخدام دالة insert:

إليك كيفية تنفيذ استعلام **INSERT** باستخدام دالة **insert**:

```
use Illuminate\Support\Facades\DB;
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

تفاصيل الاستخدام:

- **استعلامSQL** : هنا، لدينا استعلام **INSERT**يقوم بإدخال اسم و id جديدين إلى جدول **users**.
- **روابط المعاملات** :يتم تمرير القيم (اسم و id المستخدم) كمصفوفة إلى الاستعلام. تستخدم علامة الاستفهام ? كروابط للمعلمات.

ملاحظات:

١. **لا يُرجع القيم** :على عكس **select**، لا تعيد الدالة **insert**أي قيم، بل تعيد عدد الصفوف التي تم إدخالها (إذا كانت هناك حاجة لذلك، يمكن استخدام الدالة **DB::affectingStatement**.
٢. **تحقق من الخطأ** :تأكد من التحقق من الأخطاء المحتملة، مثل انتهاك قيود فريدة أو إدخالات غير صالحة، وذلك باستخدام آليات Laravel الخاصة بالتعامل مع الاستثناءات.

مثال كامل:

إليك مثالًا كاملًا لتنفيذ استعلام **INSERT**وإدخال عدة سجلات:

```
$data = [
   ['name' => 'Alice', 'email' => 'alice@example.com'],
   ['name' => 'Bob', 'email' => 'bob@example.com'],
];
foreach ($data as $user) {
   DB::insert('INSERT INTO users (name, email) VALUES (?, ?)', [$user['name'], $user['email']]);
}
```

بهذا الشكل، يمكنك إدخال بيانات جديدة في قاعدة البيانات بسهولة باستخدام طريقة insert، مما يجعل إدارة البيانات أكثر كفاءة.

(تشغيل جملةRunning an Update Statement (UPDATE

يجب استخدام الدالة (update) لتحديث السجلات الموجودة في قاعدة البيانات. يتم إرجاع عدد الصفوف المتأثرة بالتعديل بواسطة هذه الدالة.

```
use Illuminate\Support\Facades\DB;
```

```
$affected = DB::update(
   'update users set votes = 100 where name = ?',
   ['Anita']
);
```

(تشغيل جملةRunning a Delete Statement (DELETE

يجب استخدام الدالة (delete) لحذف السجلات الموجودة في قاعدة البيانات. يتم إرجاع عدد الصفوف المتأثرة بالحذف بواسطة هذه الدالة.

```
use Illuminate\Support\Facades\DB;
$deleted = DB::delete('delete from users where name = ? ',
   ['Anita']);
```

(تشغيل جملة عامة) Running a General Statement

بعض عبارات قاعدة البيانات لا تُرجع أي قيمة. بالنسبة لهذا النوع من العمليات، يمكنك استخدام الدالة statement الموجود في الـ(DB facade) :

```
DB::statement('drop table users');
```

(تشغيل جملة غير مجهزة) Running an Unprepared Statement

أحيانًا قد ترغب في تنفيذ عبارة SQL دون ربطها بأي قيم. يمكنك استخدام الدالة unprepared الموجودة في الـ (DB facade) لتحقيق ذلك:

```
DB::unprepared('update users set votes = 100 where name = "Dries"');
```

(التثبيت الضمني) Implicit Commits

عند استخدام دالتي statement و unprepared الخاصة بواجهة (DB façade) داخل المعاملات(transactions) ، يجب أن تكون حذرًا لتجنب العبارات التي تسبب عمليات "التثبيت الضمني (implicit commits) ". هذه العبارات ستؤدي إلى قيام محرك قاعدة البيانات بتثبيت المعاملة كاملة بشكل غير مباشر، مما يجعل Laravel غير مدرك لمستوى المعاملة في قاعدة البيانات. مثال على مثل هذه العبارات هو إنشاء جدول في قاعدة البيانات.

```
DB::unprepared(‘create table a (col varchar(1) null)’);
```

(استخدام اتصالات قاعدة بيانات متعددة) Using Multiple Database Connections

إذا كان تطبيقك يعرّف عدة اتصالات في ملف التكوين الخاص بك config/database.php، فيمكنك الوصول إلى كل اتصال عبر الدالة connectionالتي توفرها واجهة (DB facade).

يجب أن يكون اسم الاتصال الذي يتم تمريره إلى الدالة connection مطابقًا لأحد الاتصالات المدرجة في ملف التكوين config/database.php أو مُكوّنًا في وقت التشغيل باستخدام المُساعد config.

```
use Illuminate\Support\Facades\DB;
$users = DB::connection('sqlite')->select(/* ... */);
```

يمكنك الوصول إلى كائن PDOالأساسي الخام للاتصال باستخدام الدالة getPdo على مثيل الاتصال.

```
$pdo = DB::connection()->getPdo();
```

(الاستماع لأحداث الاستعلام) Listening for Query Events

إذا كنت ترغب في تحديد دالة إغلاق (closure) تُستدعى لكل استعلام SQL يتم تنفيذه بواسطة تطبيقك، يمكنك استخدام الدالة **listen** الخاصة بواجهة **DB**. يمكن أن تكون هذه الدالة مفيدة لتسجيل الاستعلامات أو تصحيح الأخطاء. يمكنك تسجيل دالة الإغلاق لمستمع الاستعلامات داخل الدالة **boot** لمزود الخدمة (service provider).

```
<?php
namespace App\Providers;
use Illuminate\Database\Events\QueryExecuted;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;
class AppServiceProvider extends ServiceProvider
{
   /**
    * Register any application services.
    */
   public function register(): void
   {
      // ...
   }
   /**
    * Bootstrap any application services.
    */
   public function boot(): void
   {
      DB::listen(function (QueryExecuted $query) {
         // $query->sql;       // استعلام SQL
```

```
        // $query->bindings;  // المعاملات المرتبطة
        // $query->time;      // وقت التنفيذ
        // $query->toRawSql(); // الحصول على SQL مع المعاملات
    });
  }
}
```

(مراقبة الوقت التراكمي للاستعلام) Monitoring Cumulative Query Time

أحد مشاكل عنق الزجاجة الشائع في أداء تطبيقات الويب الحديثة هو الوقت الذي تقضيه في تنفيذ استعلام قواعد البيانات.

لحسن الحظ، يمكن لـ Laravel استدعاء دالة إغلاق (closure) أو نداء (callback) تختاره عندما يستغرق الاستعلام قاعدة البيانات وقتًا طويلاً أثناء تنفيذ طلب واحد.

لحل هذه المشكلة، مرر الفترة الزمنية للاستعلام (بالملي ثانية) ودالة الإغلاق إلى الدالة **whenQueryingForLongerThan** . يمكنك استدعاء هذه الدالة في الدالة **boot** لمزود الخدمة (service provider) .

```
<?php

namespace App\Providers;
use Illuminate\Database\Connection;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;
use Illuminate\Database\Events\QueryExecuted;

class AppServiceProvider extends ServiceProvider
{
  /**
   * Register any application services.
   */
  public function register(): void
  {
    // ...
  }
   /**
   * Bootstrap any application services.
   */
  public function boot(): void
  {
    DB::whenQueryingForLongerThan(500, function (Connection $connection, QueryExecuted $event) {
      // إخطار فريق التطوير...

    });
  }
}
```

(معاملات قاعدة البيانات) Database Transactions

يمكنك استخدام الدالة **transaction** التي توفرها واجهة (DB facade) **DB** لتنفيذ مجموعة من العمليات داخل transaction قاعدة بيانات.

إذا تم إلقاء استثناء داخل دالة الإغلاق(transaction closure) ، سيتم التراجع عن الـtransaction تلقائيًا (rolled back) وإعادة رمي الاستثناء (re-thrown).

وإذا تم تنفيذ دالة الـ closure بنجاح، سيتم تثبيت الـ transaction تلقائيًا(committed) . لا تحتاج إلى القلق بشأن التراجع أو التثبيت اليدوي عند استخدام الدالة **transaction**.

```
use Illuminate\Support\Facades\DB;
DB::transaction(function () {
  DB::update('update users set votes = 1');
  DB::delete('delete from posts');
});
```

(معالجة التعارضات) Handling Deadlocks

يُمرر الى الدالة **transaction** قيمة اختيارية ثانية تحدد عدد المرات التي يجب فيها إعادة المحاولة في حالة حدوث تعارض (deadlock) أثناء الـ transaction. بمجرد استنفاد هذه المحاولات، سيتم إلقاء استثناء(exception) .

```
use Illuminate\Support\Facades\DB;
```

```
DB::transaction(function () {
   DB::update('update users set votes = 1');
   DB::delete('delete from posts');
}, 5); // محاولات 5
```

(الاستخدام اليدوي للمعاملات) Manually Using Transactions

إذا كنت ترغب في بدء الـ transaction يدويًا والحصول على التحكم الكامل في التراجع (rollback) والتثبيت(commit) ، يمكنك استخدام الدالة **beginTransaction** التي توفرها الـ DB facade.

```
use Illuminate\Support\Facades\DB;
DB::beginTransaction(); // بدء المعاملة
```

تستطيع التراجع عن المعاملة باستخدام الدالة rollBack

```
DB::rollBack(); // التراجع عن المعاملة
```

تستطيع تثبيت المعاملة باستخدام commit

```
DB::commit(); // تثبيت المعاملة
```

(الاتصال بواجهة سطر أوامر قاعدة البيانات) Connecting to the Database CLI

إذا كنت ترغب في الاتصال بواجهة الأوامر (CLI) الخاصة بقاعدة بياناتك، يمكنك استخدام الامر التالي:

```
php artisan db
```

إذا لزم الأمر، يمكنك تحديد اسم اتصال قاعدة البيانات للارتباط باتصال قاعدة بيانات غير الاتصال الافتراضي، عن طريق تمرير اسم الاتصال كخيار إضافي.

```
php artisan db mysql
```

(فحص قواعد البيانات) Inspecting Your Databases

باستخدام الامر db:show والامر db:table، يمكنك الحصول على نظرة عامة او ملخصة حول قاعدة بياناتك والجداول المرتبطة بها. لعرض نظرة عامة على قاعدة بياناتك، بما في ذلك حجمها، نوعها، عدد الاتصالات المفتوحة، وملخص للجداول، يمكنك استخدام الأمر **db**

```
php artisan db:table
```

يمكنك تحديد اتصال قاعدة البيانات الذي يجب فحصه عن طريق تقديم اسم اتصال قاعدة البيانات إلى الأمر باستخدام الخيار **--database**

```
php artisan db:table --database=pgsql
```

إذا كنت ترغب في تضمين عدد الصفوف في الجداول وتفاصيل العروض (views) في مخرجات الأمر، يمكنك استخدام الخيارين **--counts** و **--views**على التوالي. قد يكون استرجاع عدد الصفوف وتفاصيل العروض بطيئًا في قواعد البيانات الكبيرة.

```
php artisan db:table --counts --views
```

يمكنك استخدام دوال أخرى لتفحص قواعد البيانات.

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

$tables = Schema::getTables();        // الحصول على جميع الجداول
$views = Schema::getViews();          // الحصول على جميع العروض
$columns = Schema::getColumns('users'); // الحصول على أعمدة جدول معين
$indexes = Schema::getIndexes('users'); // الحصول على فهارس جدول معين
$foreignKeys = Schema::getForeignKeys('users'); // الحصول على المفاتيح الأجنبية
```

إذا كنت ترغب في فحص اتصال قاعدة بيانات غير الاتصال الافتراضي لتطبيقك، يمكنك استخدام الدالة **connection**.

```
$columns = Schema::connection('sqlite')->getColumns('users');
```

(نظرة عامة على الجدول) Table Overview

إذا كنت ترغب في الحصول على نظرة عامة عن جدول معين داخل قاعدة بياناتك، يمكنك تنفيذ أمر **db** الخاص بـ **Artisan.** يوفر هذا الأمر نظرة عامة شاملة عن الجدول، بما في ذلك أعمدته، أنواعه، خصائصه، المفاتيح، والفهارس(indexes) .

```
php artisan db:table users
```

(مراقبة قواعد البيانات) Monitoring Your Databases

باستخدام الامر artisan التالي db:monitor يمكنك توجيه Laravel لإطلاق حدث **Illuminate\Database\Events\DatabaseBusy** إذا كانت قاعدة البيانات تتعامل مع عدد كثير من الاتصالات المفتوحة.

للبدء، يجب عليك جدولة أمر db:monitor ليتم تشغيله كل دقيقة. يقبل الأمر أسماء اتصالات قاعدة البيانات التي تم تهيئتها والتي ترغب في مراقبتها، بالإضافة إلى الحد الأقصى لعدد الاتصالات المفتوحة التي ينبغي السماح بها قبل إطلاق الحدث.

```
php artisan db:monitor --databases=mysql,pgsql --max=100
```

جدولة هذا الأمر وحدها لا تكفي لتفعيل إشعار ينبهك حول عدد الاتصالات المفتوحة. عندما يواجه الأمر قاعدة بيانات تحتوي على عدد اتصالات مفتوحة يتجاوز الحد المسموح به، سيتم إطلاق حدث **DatabaseBusy**.

يجب عليك الاستماع إلى هذا الحدث داخل **AppServiceProvider**الخاص بتطبيقك من أجل إرسال إشعار إليك أو إلى فريق التطوير الخاص بك. يمكنك القيام بذلك عن طريق تسجيل مستمع الحدث **DatabaseBusy** وإرسال الإشعارات المناسبة عندما يتم تجاوز الحد المسموح به لعدد الاتصالات المفتوحة.

```
use App\Notifications\DatabaseApproachingMaxConnections;
use Illuminate\Database\Events\DatabaseBusy;
```

```
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;
/**
 * Bootstrap any application services.
 */
public function boot(): void
{
   Event::listen(function (DatabaseBusy $event) {
      Notification::route('mail', 'dev@example.com')
            ->notify(new DatabaseApproachingMaxConnections(
               $event->connectionName,
               $event->connections
            ));
   });
}
```

(منشئ استعلامات قاعدة البيانات) Database: Query Builder

(مقدمة) Introduction

يوفر مُنشئ استعلامات قاعدة البيانات في Laravel واجهة سهلة وسلسة لإنشاء وتشغيل استعلامات قاعدة البيانات. يمكن استخدامه لتنفيذ معظم العمليات على قاعدة البيانات في تطبيقك، ويعمل بشكل مثالي مع جميع أنظمة قواعد البيانات التي يدعمها Laravel .

يستخدم مُنشئ الاستعلامات في Laravel ربط معلمات **PDO** لحماية تطبيقك من هجمات حقن SQL (SQL injection) . ليس هناك حاجة لتنظيف أو تصفية السلاسل النصية التي تُمرر إلى مُنشئ الاستعلامات كارتباطات استعلام(query bindings) .

(تشغيل استعلامات قاعدة البيانات) Running Database Queries

(استرجاع كل الصفوف من الجدول) Retrieving All Rows From a Table

يمكنك استخدام الدالة **table** التي توفرها واجهة **DB** (DB facade) لبدء استعلام. تقوم الدالة **table** بإرجاع كائن مُنشئ استعلامات سلس(fluent query builder) للجدول المُحدد، مما يتيح لك إضافة المزيد من القيود إلى الاستعلام بشكل متسلسل، ثم في النهاية يتم استرجاع نتائج الاستعلام باستخدام الدالة **get**.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Support\Facades\DB;
use Illuminate\View\View;
class UserController extends Controller
{
   /**
    * Show a list of all of the application's users.
    */
   public function index(): View
   {
      $users = DB::table('users')->get(); // استرجاع جميع المستخدمين
      return view('user.index', ['users' => $users]);
   }
}
```

تُرجع الدالة **get** كائنًا من نوع **Illuminate\Support\Collection** يحتوي على نتائج الاستعلام، حيث تكون كل نتيجة عبارة عن مثيل لكائن **stdClass** فيPHP . يمكنك الوصول إلى قيمة كل عمود من خلال الوصول إلى العمود كخاصية (property) للكائن.

```
use Illuminate\Support\Facades\DB;
$users = DB::table('users')->get();
foreach ($users as $user) {
   echo $user->name; // الوصول إلى اسم المستخدم
}
```

(استرجاع صف/عمود واحد من الجدول) Retrieving a Single Row / Column From a Table

إذا كنت بحاجة إلى استرجاع صف واحد فقط من جدول قاعدة البيانات، يمكنك استخدام الدالة **first** الخاصة بواجهة **DB** (DB facade). ستقوم هذه الدالة بإرجاع كائن **stdClass** واحد.

```
$user = DB::table('users')->where('name', 'John')->first(); // الحصول على أول مستخدم باسم John
return $user->email; // إرجاع البريد الإلكتروني
```

إذا كنت ترغب في استرجاع صف واحد من جدول قاعدة البيانات، ولكن تريد إلقاء استثناء **Illuminate\Database\RecordNotFoundException** إذا لم يتم العثور على صف مطابق، يمكنك استخدام الدالة **firstOrFail.**

إذا لم يتم التقاط استثناء **RecordNotFoundException**، فسيتم تلقائيًا إرسال استجابة **404 HTTP** إلى العميل.

```
$user = DB::table('users')->where('name', 'John')->firstOrFail(); // إلقاء استثناء إذا لم يوجد
```

إذا كنت لا تحتاج إلى صف كامل، يمكنك استخراج قيمة واحدة من سجل باستخدام الدالة **value**. ستقوم هذه الدالة بإرجاع قيمة العمود مباشرة.

```
$email = DB::table('users')->where('name', 'John')->value('email'); // الحصول على البريد الإلكتروني مباشرة
```

اذا كنت ترغب باسترجاع سجل باستخدام عمود الـ id (المفتاح الرئيسي) يمكنك استخدام الدالة find وتمرير id لها.

```
$user = DB::table('users')->find(3); // البحث باستخدام المفتاح الأساسي
```

(استرجاع قائمة بقيم العمود) Retrieving a List of Column Values

إذا كنت ترغب في الحصول على كائن من نوع Illuminate\Support\Collection يحتوي على قيم عمود واحد، يمكنك استخدام الدالة pluck. في هذا المثال، سنسترجع مجموعة من القاب المستخدمين:

```
use Illuminate\Support\Facades\DB;
$titles = DB::table('users')->pluck('title'); // الحصول على جميع الألقاب
foreach ($titles as $title) {
    echo $title;
}
```

يمكنك تحديد العمود الذي يجب أن تستخدمه المجموعة الناتجة كمفاتيح لها من خلال توفير معامل ثانية للدالة pluck:

```
$titles = DB::table('users')->pluck('title', 'name'); // المفاتيح هي الأسماء والقيم هي الألقاب
foreach ($titles as $name => $title) {
    echo $title;
}
```

(تقسيم النتائج) Chunking Results

إذا كنت بحاجة إلى العمل مع آلاف من سجلات قاعدة البيانات، ففكر في استخدام الدالة chunk التي توفرها واجهة قاعدة البيانات. تسترد هذه الدالة جزءًا صغيرًا من النتائج في كل مرة وتمرره في إغلاق للمعالجة. على سبيل المثال، دعنا نسترد جدول المستخدمين بالكامل في أجزاء من ١٠٠ سجل في المرة الواحدة:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\DB;
DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {
    foreach ($users as $user) {// معالجة ١٠٠ سجل في كل مرة
        // ...
    }
});
```

يمكنك إيقاف معالجة المزيد من القطع عن طريق إرجاع false من الإغلاق:

```
DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {
    // Process the records...
    return false; // إيقاف التقسيم
});
```

إذا كنت تقوم بتحديث سجلات قاعدة البيانات أثناء تقسيم النتائج، فقد تتغير نتائج تقسيم البيانات بطرق غير متوقعة. إذا كنت تخطط لتحديث السجلات المستردة أثناء تقسيم البيانات، فمن الأفضل دائمًا استخدام الدالة chunkById بدلاً من ذلك. ستقوم هذه الدالة بتقسيم النتائج تلقائيًا استنادًا إلى المفتاح الأساسي للسجل:

```
DB::table('users')->where('active', false)
    ->chunkById(100, function (Collection $users) {
        foreach ($users as $user) {
            DB::table('users')
                ->where('id', $user->id)
                ->update(['active' => true]); // تحديث السجلات
        }
    });
```

(تدفق النتائج بتكاسل) Streaming Results Lazily

تعمل الدالة lazy بشكل مشابه للدالة chunk بمعنى أنها تنفذ الاستعلام في أجزاء. ومع ذلك، بدلاً من تمرير كل جزء إلى callback ، تقوم الدالة lazy بإرجاع LazyCollection، مما يتيح لك التفاعل مع النتائج كتدفق واحد:

```
use Illuminate\Support\Facades\DB;
DB::table('users')->orderBy('id')->lazy()->each(function (object $user) {
    // معالجة كل سجل بشكل فردي
});
```

إذا كنت تخطط لتحديث السجلات التي تم استرجاعها أثناء المرور عليها، فمن الأفضل استخدام دوال مثل lazyById أو lazyByIdDesc بدلاً من ذلك. تقوم هذه الدوال بتقسيم النتائج تلقائيًا إلى صفحات استنادًا إلى المفتاح الأساسي للسجل.

```
DB::table('users')->where('active', false)
    ->lazyById()->each(function (object $user) {
```

```
    DB::table('users')
        ->where('id', $user->id)
        ->update(['active' => true]); // تحديث السجلات
});
```

(التجميعات) Aggregates

يوفر مُنشئ الاستعلام أيضًا مجموعة متنوعة من الدوال لاسترجاع القيم المجمعة مثل count، و max، و min، و avg، و sum. يمكنك استدعاء أي من هذه الدوال بعد بناء استعلامك.

```
use Illuminate\Support\Facades\DB;
$users = DB::table('users')->count();  // عدد المستخدمين
$price = DB::table('orders')->max('price'); // أعلى سعر
```

بالطبع، يمكنك دمج هذه الدوال مع جمل أخرى لضبط كيفية حساب القيمة المجمعة بدقة.

```
$price = DB::table('orders')
            ->where('finalized', 1)
            ->avg('price'); // متوسط السعر للطلبات النهائية
```

(تحديد وجود السجلات) Determining if Records Exist

بدلاً من استخدام الدالة count لتحديد ما إذا كانت هناك أي سجلات تطابق قيود استعلامك، يمكنك استخدام الدالة exists والدالة doesntExist.

```
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // يوجد طلبات نهائية
}
if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
    // لا يوجد طلبات نهائية
}
```

(جملSELECT) Select Statements

(تحديد جملةSELECT) Specifying a Select Clause

في بعض الحالات لا تريد استرجاع بيانات كل أعمدة الجدول، باستخدام الدالة select وعمل تخصيص لعبارتها يمكنك استرجاع الاعمدة التي تريد كما في المثال التالي:

```
use Illuminate\Support\Facades\DB;
$users = DB::table('users')
        ->select('name', 'email as user_email') // تحديد أعمدة معينة مع
        ->get();
```

تسمح لك دالة distinctباجبار الاستعلام (query) على إرجاع نتائج فريدة (غير مكررة) فقط.

```
$users = DB::table('users')->distinct()->get(); // نتائج فريدة بدون تكرار
```

إذا كان لديك بالفعل كائن منشئ استعلامات(query builder instance) ، وترغب في إضافة عمود إلى جملة selectالحالية الخاصة به، يمكنك استخدام دالة addSelect.

```
$query = DB::table('users')->select('name');
$users = $query->addSelect('age')->get(); // إضافة عمود العمر
```

(التعبيرات الخام) Raw Expressions

أحيانًا قد تحتاج إلى إدراج سلسلة نصية خام (arbitrary string) في استعلام ما. لإنشاء تعبير نصي خام(raw string expression) ، يمكنك استخدام دالة rawالتي توفرها الـ (DB facade).

```
$users = DB::table('users')
        ->select(DB::raw('count(*) as user_count, status')) // استخدام تعبير خام
        ->where('status', '<>', 1)
        ->groupBy('status')
        ->get();
```

ملاحظة : سيتم حقن التعبيرات الخام (Raw statements) في الاستعلام كنصوص(strings) ، لذا يجب أن تكون حذراً للغاية لتجنب إنشاء ثغرات أمنية من النوع (SQL injection vulnerabilities).

(الدوال الخام) Raw Methods

بدلاً من استخدام دالة DB::raw، يمكنك أيضًا استخدام الدوال التالية لإدراج تعبير خام (raw expression) في أجزاء مختلفة من استعلامك. تذكّر، لا يمكن لـ Laravel أن تضمن حماية أي استعلام يستخدم تعبيرات خام من ثغرات الـ(SQL injection vulnerabilities) .

selectRaw

يمكن استخدام دالة selectRaw كبديل لـ addSelect(DB::raw(...)). تقبل هذه الدالة مصفوفة اختيارية من معاملات الربط (bindings) كوسيط (argument) ثانٍ لها.

```
$orders = DB::table('orders')
        ->selectRaw('price * ? as price_with_tax', [1.0825]) // حساب السعر مع الضريبة
```

```
        ->get();
```

whereRaw / orWhereRaw

يمكن استخدام دالتي whereRawو orWhereRawلحقن جملة "where" خام في استعلامك. تقبل هاتان الدالتان مصفوفة اختيارية من معاملات الربط (bindings)كوسيط (argument) ثانٍ لهما.

```
$orders = DB::table('orders')
        ->whereRaw('price > IF(state = "TX", ?, 100)', [200]) // شرط خام مع معامل
        ->get();
```

havingRaw / orHavingRaw

يمكن استخدام دالتي havingRawو orHavingRawلتوفير سلسلة نصية خام كقيمة لجملة ."having" تقبل هاتان الدالتان مصفوفة اختيارية من معاملات الربط (bindings) كوسيط (argument) ثانٍ لهما.

```
$orders = DB::table('orders')
        ->select('department', DB::raw('SUM(price) as total_sales'))
        ->groupBy('department')
        ->havingRaw('SUM(price) > ?', [2500]) // شرط خام على having
        ->get();
```

orderByRaw

يمكن استخدام دالة orderByRawلتوفير سلسلة نصية خام كقيمة لجملة."order by"

```
$orders = DB::table('orders')
        ->orderByRaw('updated_at - created_at DESC') // ترتيب حسب تعبير خام
        ->get();
```

groupByRaw

يمكن استخدام دالة groupByRaw لتوفير سلسلة نصية خام كقيمة لجملة "group by" .

```
$orders = DB::table('orders')
        ->select('city', 'state')
        ->groupByRaw('city, state') // تجميع حسب تعبير خام
        ->get();
```

Joins (الربط)

Inner Join Clause (جملة الربط الداخلي)

يمكن أيضًا استخدام منشئ الاستعلامات (query builder) لإضافة جمل الربط (join clauses) إلى استعلاماتك.

لتنفيذ ربط داخلي أساسي("inner join") ، يمكنك استخدام دالة joinعلى كائن منشئ الاستعلامات. الوسيط (argument) الأول الذي يتم تمريره إلى دالة joinهو اسم الجدول الذي تحتاج إلى الربط به، بينما تحدد الوسائط المتبقية قيود الأعمدة اللازمة لعملية الربط.

يمكنك حتى ربط عدة جداول في استعلام واحد.

```
use Illuminate\Support\Facades\DB;
$users = DB::table('users')
        ->join('contacts', 'users.id', '=', 'contacts.user_id') // ربط جدول contacts
        ->join('orders', 'users.id', '=', 'orders.user_id')     // ربط جدول orders
        ->select('users.*', 'contacts.phone', 'orders.price')   // تحديد الأعمدة
        ->get();
```

Left Join / Right Join Clause (جملة الربط الأيسر/الأيمن)

إذا كنت ترغب في تنفيذ "ربط أيسر (left join) "أو "ربط أيمن (right join) "بدلاً من "الربط الداخلي(inner join) "، فاستخدم دالتي leftJoin أو rightJoin. هاتان الدالتان لهما نفس بنية (signature) دالة .join

```
$users = DB::table('users')
        ->leftJoin('posts', 'users.id', '=', 'posts.user_id') // ربط أيسر
        ->get();
$users = DB::table('users')
        ->rightJoin('posts', 'users.id', '=', 'posts.user_id') // ربط أيمن
        ->get();
```

Cross Join Clause (جملة الربط المتقاطع)

يمكنك استخدام دالة crossJoinلتنفيذ "ربط متقاطع .(cross join) "تقوم عمليات الربط المتقاطع بإنشاء حاصل ضرب ديكارتي (cartesian product) بين الجدول الأول والجدول المربوط.

```
$sizes = DB::table('sizes')
        ->crossJoin('colors') // ربط متقاطع
        ->get();
```

Advanced Join Clauses (جمل الربط المتقدمة)

يمكنك أيضًا تحديد جمل ربط (join clauses) أكثر تقدمًا. للبدء، قم بتمرير دالة مجهولة (closure) كوسيط (argument) ثانٍ لدالة join. ستستقبل هذه الدالة المجهولة كائنًا من نوع Illuminate\Database\Query\JoinClause، والذي يسمح لك بتحديد القيود (constraints) على جملة"join" .

```
DB::table('users')
    ->join('contacts', function (JoinClause $join) {
        $join->on('users.id', '=', 'contacts.user_id')->orOn(/* ... */); // شروط متقدمة
    })
    ->get();
```

إذا كنت ترغب في استخدام جملة "where" على عمليات الربط (joins) الخاصة بك، يمكنك استخدام دالتي where وorWhere التي يوفرها كائن JoinClause. بدلاً من مقارنة عمودين، ستقوم هاتان الدالتان بمقارنة العمود بقيمة(value) .

```
DB::table('users')
    ->join('contacts', function (JoinClause $join) {
        $join->on('users.id', '=', 'contacts.user_id')
            ->where('contacts.user_id', '>', 5); // شرط where على الربط
    })
    ->get();
```

(ربط الاستعلامات الفرعية) Subquery Joins

يمكنك استخدام دوال joinSub، وleftJoinSub، و rightJoinSubلربط استعلام باستعلام فرعي(subquery) . تستقبل كل من هذه الدوال ثلاثة وسائط (arguments): الاستعلام الفرعي، واسمه المستعار للجدول(table alias) ، ودالة مجهولة (closure) تحدد الأعمدة المرتبطة.

في هذا المثال، سنقوم باسترداد مجموعة من المستخدمين حيث يحتوي كل سجل مستخدم أيضًا على الطابع الزمني created_atلأحدث تدوينة نشرها هذا المستخدم:

```
$latestPosts = DB::table('posts')
            ->select('user_id', DB::raw('MAX(created_at) as last_post_created_at'))
            ->where('is_published', true)
            ->groupBy('user_id'); // استعلام فرعي

$users = DB::table('users')
    ->joinSub($latestPosts, 'latest_posts', function (JoinClause $join) {
        $join->on('users.id', '=', 'latest_posts.user_id'); // ربط بالاستعلام الفرعي
    })->get();
```

(الربط الجانبي) Lateral Joins

تدعم قواعد البيانات التالية حاليًا الربط الجانبيPostgreSQL (Lateral Joins): ، و MySQL (الإصدار ٨,٠,١٤ فما فوق)، وSQL Server .

يمكنك استخدام دالتي joinLateral و leftJoinLateral لتنفيذ "ربط جانبي (lateral join) "مع استعلام فرعي(subquery) .

تستقبل كل من هاتين الدالتين وسيطين(arguments) : الاستعلام الفرعي واسمه المستعار للجدول(table alias) .

يجب تحديد شرط (أو شروط) الربط داخل جملة whereالخاصة بالاستعلام الفرعي المُعطى. يتم تقييم الربط الجانبي لكل صف على حدة، ويمكنه الإشارة إلى أعمدة خارج الاستعلام الفرعي.

في هذا المثال، سنقوم باسترداد مجموعة من المستخدمين بالإضافة إلى أحدث ثلاث تدوينات لكل مستخدم. يمكن لكل مستخدم أن يُنتج ما يصل إلى ثلاثة صفوف في مجموعة النتائج: صف واحد لكل تدوينة من تدويناته الأحدث. يتم تحديد شرط الربط (join condition) باستخدام جملة whereColumnداخل الاستعلام الفرعي، مع الإشارة إلى صف المستخدم الحالي.

```
$latestPosts = DB::table('posts')
            ->select('id as post_id', 'title as post_title', 'created_at as post_created_at')
            ->whereColumn('user_id', 'users.id') // شرط الربط الجانبي
            ->orderBy('created_at', 'desc')
            ->limit(3); // الاستعلام الفرعي

$users = DB::table('users')
        ->joinLateral($latestPosts, 'latest_posts') // ربط جانبي
        ->get();
```

(الدمج) Unions

يوفر منشئ الاستعلامات (query builder) أيضًا دالة ملائمة لدمج (union) استعلامين أو أكثر معًا. على سبيل المثال، يمكنك إنشاء استعلام أولي واستخدام دالة unionلدمجه مع استعلامات أخرى.

```
use Illuminate\Support\Facades\DB;
$first = DB::table('users')
        ->whereNull('first_name'); // الاستعلام الأول
$users = DB::table('users')
```

```
        ->whereNull('last_name')
        ->union($first) // دمج الاستعلامات
        ->get();
```

بالإضافة إلى دالة union، يوفر منشئ الاستعلامات الدالة unionAll. الاستعلامات التي يتم دمجها باستخدام الدالة unionAll لن تتم إزالة نتائجها المكررة. الدالة unionAll لها نفس بنية الدالة union.

Basic Where Clauses (جمل WHERE الأساسية)

Where Clauses (WHERE جمل)

يمكنك استخدام الدالة where الخاصة بمنشئ الاستعلامات لإضافة جمل العبارة where إلى الاستعلام. يتطلب الاستدعاء الأساسي للدالة where ثلاثة وسائط (arguments):

١. الوسيط الأول: هو اسم العمود.
٢. الوسيط الثاني: هو عامل المقارنة (operator)، والذي يمكن أن يكون أيًا من عوامل المقارنة المدعومة في قاعدة البيانات.
٣. الوسيط الثالث: هو القيمة التي سيتم مقارنتها بقيمة العمود.

على سبيل المثال، الاستعلام التالي يقوم باسترجاع المستخدمين الذين تكون قيمة عمود votes لديهم تساوي ١٠٠ وقيمة عمود age أكبر من ٣٥:

```
$users = DB::table('users') // تجميع النتائج
            ->where('votes', '=', 100)
            ->where('age', '>', 35)
            ->get();
```

للتسهيل، إذا أردت التحقق من أن قيمة عمود ما بإنها تساوي قيمة معينة، يمكنك تمرير هذه القيمة مباشرة كوسيط (argument) ثانٍ للدالة where. سيفترض Laravel أنك تريد استخدام عامل المقارنة =.

```
$users = DB::table('users')->where('votes', 100)->get(); // اختصار للمساواة
```

كما ذُكر سابقًا، يمكنك استخدام أي عامل مقارنة (operator) تدعمه نظام قاعدة البيانات الخاصة بك.

```
$users = DB::table('users')
            ->where('votes', '>=', 100)
            ->get();
$users = DB::table('users')
            ->where('votes', '<>', 100)
            ->get();
$users = DB::table('users')
            ->where('name', 'like', 'T%')
            ->get();
```

يمكنك أيضًا تمرير مصفوفة من الشروط إلى للدالة where. يجب أن يكون كل عنصر في هذه المصفوفة عبارة عن مصفوفة أخرى تحتوي على الوسائط الثلاثة التي يتم تمريرها عادةً إلى الدالة where.

```
$users = DB::table('users')->where([
   ['status', '=', '1'],
   ['subscribed', '<>', '1'],
])->get();
```

لا تدعم PDO ربط بتسمية الأعمدة(binding column names) . لذلك، يجب ألا تسمح أبدًا لمدخلات المستخدم بأن تحدد أسماء الأعمدة التي تشير إليها استعلاماتك، بما في ذلك أعمدة order by.

Or Where Clauses (OR WHERE جمل)

عند ربط استدعاءات الدالة where الخاصة بمنشئ الاستعلامات معًا بشكل متسلسل، سيتم ضم جمل العبارة where باستخدام العامل and.

ولكن، يمكنك استخدام الدالة orWhere لضم جملة شرطية إلى الاستعلام باستخدام العامل or. تقبل دالة orWhere نفس الوسائط (arguments) التي تقبلها الدالة where.

```
$users = DB::table('users')
            ->where('votes', '>', 100)
            ->orWhere('name', 'John')
            ->get();
```

إذا كنت بحاجة إلى تجميع شرط or داخل أقواس، يمكنك تمرير دالة مجهولة (closure) كوسيط (argument) أول للدالة orWhere.

```
$users = DB::table('users')
        ->where('votes', '>', 100)
        ->orWhere(function (Builder $query) {
            $query->where('name', 'Abigail')
                ->where('votes', '>', 50);
        })
```

```
        ->get();
```

المثال السابق سوف ينتج عنه عبارة SQL التالية:

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

يجب عليك دائمًا تجميع استدعاءات orWhere لتجنب السلوك غير المتوقع عند تطبيق النطاقات العامة(global scopes) .

Where Not Clauses (WHERE NOT جمل)

يمكن استخدام دالتي whereNot و orWhereNot لنفي مجموعة معينة من قيود الاستعلام. على سبيل المثال، يستبعد الاستعلام التالي المنتجات التي هي قيد التصفية (on clearance) أو التي يقل سعرها عن عشرة:

```
$products = DB::table('products')
        ->whereNot(function (Builder $query) {
            $query->where('clearance', true)
                ->orWhere('price', '<', 10);
        })
        ->get();
```

Where Any / All / None Clauses (WHERE ANY/ALL/NONE جمل)

أحيانًا قد تحتاج إلى تطبيق نفس قيود الاستعلام على عدة أعمدة. على سبيل المثال، قد ترغب في استرجاع جميع السجلات لقائمة معينة من الاعمدة تمرر اسماءها الى الدالة whereAny تشبه LIKE قيمها قيمة معينة.

```
$users = DB::table('users')
        ->where('active', true)
        ->whereAny([
            'name',
            'email',
            'phone',
        ], 'like', 'Example%')
        ->get();
```

الاستعلام السابق سوف ينتج عنه عبارة الـ SQL التالية:

```
SELECT *
FROM users
WHERE active = true AND (
    name LIKE 'Example%' OR
    email LIKE 'Example%' OR
    phone LIKE 'Example%'
)
```

---

lecture 07 Eloquent ORM الملزمة الثانية

Eloquent ORM:

Introduction (مقدمة)

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 (إنشاء فئات النماذج)

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 // لإنشاء نموذج جديد make:model Artisan يمكنك استخدام أمر
```

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

```
php artisan make:model Flight –migration // إذا كنت ترغب في إنشاء جدول قاعدة البيانات عند إنشاء الـ model
```

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:

```
# إنشاء نموذج وفئة FlightFactory...

php artisan make:model Flight –factory      // إنشاء مصنع
php artisan make:model Flight -f            // هذه اختصارها

# إنشاء نموذج وفئة FlightSeeder...

php artisan make:model Flight –seed         // إنشاء بذرة
php artisan make:model Flight -s            // هذه اختصارها
```

```
# إنشاء نموذج وفئة FlightController...
php artisan make:model Flight –controller    // إنشاء وحدة تحكم
php artisan make:model Flight -c             // هذه اختصارها
# إنشاء نموذج، وفئة FlightController كنموذج مورد (Resource)، وفئات Form Request...
php artisan make:model Flight -crR
# إنشاء نموذج وفئة FlightPolicy...
php artisan make:model Flight –policy  //
# إنشاء نموذج وهجرة، seeder ،factory، و controller...
php artisan make:model Flight -mfsc  // إنشاء نموذج، ترحيل، مصنع، بذرة، ووحدة تحكم
# اختصار لإنشاء نموذج، هجرة، controller ،policy، seeder، factory، و form requests...
php artisan make:model Flight --all
php artisan make:model Flight -a
# إنشاء نموذج Pivot (وسيط)...
php artisan make:model Member --pivot
php artisan make:model Member -p
```

فحص النماذج - Inspecting Models

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
```

اتفاقيات نموذجEloquent - Eloquent Model 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
{
   // ...
}
```

(أسماء الجداول) Table Names

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.

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
{
   /**
    * الجدول المرتبط بالنموذج.
    *
    * @var string
    */
   protected $table = 'my_flights';
}
```

(المفاتيح الأساسية) Primary Keys

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.
     *المفتاح الأساسي المرتبط بالجدول.
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

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
{
    /**
     * يشير إلى ما إذا كان المفتاح الأساسي يزداد تلقائيًا أم لا.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

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
{
    /**
     * نوع بيانات معرف المفتاح الأساسي.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

(المفاتيح الأساسية المركبة) Composite" Primary Keys"

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.

(مفاتيحULID و UUID)UUID and ULID Keys

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 ٣٦ حرفًا.

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"
```

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.

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;
/**
 * إنشاءUUID جديد للنموذج
 */
public function newUniqueId(): string
{
    return (string) Uuid::uuid4();
}
/**
 الحصول على الأعمدة التي يجب أن تستقبل معرفًا فريدًا. *
 *
 * @return array<int, string>
 */
public function uniqueIds(): array
{
    return ['id', 'discount_code'];
}
```

إذا كنت ترغب في استخدام "معرفات ULID" بدلاً من معرفات UUID. معرفات ULID تشبه معرفات UUID؛ ومع ذلك، فهي تتكون من ٢٦ 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"
```

(الطوابع الزمنية)Timestamps

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.
     يشير إلى ما إذا كان النموذج يجب أن يكون له طوابع زمنية. *
     * @var bool
     */
    public $timestamps = false;
}
```

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
{
```

```
    /**
     * تنسيق التخزين لأعمدة تاريخ النموذج.
     * @var string
     */
    protected $dateFormat = 'U';
}
```

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';
}
```

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'));
```

(اتصال قواعد البيانات) Database Connections

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
{
    /**
     * اتصال قاعدة البيانات الذي يجب أن يستخدمه النموذج.
     *
     * @var string
     */
    protected $connection = 'mysql';
}
```

(قيمة الصفة الافتراضية) Default Attribute Values

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
{
    /**
     * القيم الافتراضية للنموذج للصفات او السمات.
     *
     * @var array
     */
    protected $attributes = [
        'options' => '[]',
        'delayed' => false,
    ];
}
```

تكوين صرامة Eloquent (Configuring Eloquent Strictness)

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;
تهيئة أي خدمات للتطبيق. **/
 * Bootstrap any application services.
 */
public function boot(): void
{
   Model::preventLazyLoading(! $this->app->isProduction());//يمنع التحميل الكسول في البيئات غير الإنتاجية
}
```

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());
```

استرداد النماذج(Retrieving Models)

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;
}
```

استعلامات البناء (Building Queries)

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();
```

تحديث النماذج(Refreshing Models)

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();
```

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"
```

المجموعات(Collections)

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.
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;// النتائج تُرجع كائن Collection مثل دوال مساعدة معreject، إلخ each،
});
```

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

تجزئة النتائج إلى أجزاء (Chunking Results)

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.
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) {// يسترجع السجلات في أجزاء لتجنب استنفاد الذاكرة.
        // ...
    }
});
```

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 .
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');
```

التقسيم إلى أجزاء باستخدام المجموعات الكسولة (Chunking Using Lazy Collections)

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) {
    // ...
}
```

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]);
```

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

المؤشرات (Cursors)

على غرار الدالة lazy، يمكن استخدام الدالة cursor لتقليل استهلاك تطبيقك للذاكرة بشكل كبير عند التكرار عبر عشرات الآلاف من سجلات نموذج Eloquent.
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.
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.
داخليًا، تستخدم الدالة cursor مولدات PHP لتنفيذ هذه الوظيفة:

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

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;
}
```

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.

(استعلامات فرعية متقدمة)Advanced Subqueries

(يختار الاستعلام الفرعي)Subquery Selects

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.

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')  // استعلام فرعي في SELECT
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
])->get();
```

(طلب الاستعلام الفرعي) Subquery Ordering

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') //// استعلام فرعي في ORDER BY
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

Retrieving Single Models / Aggregates (استرداد النماذج الفردية / المجاميع)

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); // استرجاع نموذج باستخدام المفتاح الأساسي...
// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();// استرجاع أول نموذج مطابق لشروط الاستعلام...
// Alternative to retrieving the first model matching the query constraints...
$flight = Flight::firstWhere('active', 1); // طريقة بديلة لاسترجاع أول نموذج مطابق لشروط الاستعلام...
```

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 () {
    // ...
});
```

(لم يتم العثور على استثناء) Not Found Exceptions

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();
```

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);
});
```

(استرداد او انشاء نماذج) Retrieving or Creating Models

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).
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([ // استرجع رحلة بالاسم أو أنشئها إذا لم تكن موجودة...
   'name' => 'London to Paris'
]);
// Retrieve flight by name or create it with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrCreate(// استرجع رحلة بالاسم أو أنشئها مع السمات name, delayed, و arrival_time...
   ['name' => 'London to Paris'],
   ['delayed' => 1, 'arrival_time' => '11:30']
);
// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrNew([// استرجع رحلة بالاسم أو أنشئ كائن جديد دون حفظه في قاعدة البيانات...
   'name' => 'London to Paris'
]);
// Retrieve flight by name or instantiate with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrNew(// استرجع رحلة بالاسم أو أنشئ كائن جديد مع السمات name, delayed, و arrival_time...
   ['name' => 'Tokyo to Sydney'],
   ['delayed' => 1, 'arrival_time' => '11:30']
);
```

(استرداد المجاميع) Retrieving Aggregates

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');
```

(ادخال وتحديث النماذج) Inserting and Updating Models

(ادخالات) Inserts

بالطبع، عند استخدام Eloquent ، 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
{
   /**    تخزين رحلة جديدة في قاعدة البيانات.
    * Store a new flight in the database.
    */
   public function store(Request $request): RedirectResponse
   {
      // Validate the request...        // (مفقود في هذا المثال) التحقق من صحة البيانات...
       $flight = new Flight;        // إنشاء كائن جديد من نموذج الرحلة
       $flight->name = $request->name;        // تعيين اسم الرحلة من بيانات الطلب
       $flight->save();        // حفظ الرحلة في قاعدة البيانات
       return redirect('/flights');        // إعادة توجيه إلى صفحة الرحلات
   }
}
```

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.
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',
]);
```

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؟

التعيين الجماعي(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.

(تحديثات)Updates

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();
```

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.
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]
);
```

(تحديثات جماعية)Mass Updates

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]);
```

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.
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.

(فحص تغييرات الصفات) Examining Attribute Changes

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.
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) ".
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
```

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
```

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...
```

(التعيين الجماعي)Mass Assignment

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',
]);
```

قبل استخدام الدالة **create**فيEloquent ، 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.

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.

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'];
}
```

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']);
```

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']);
```

(التعيين الجماعي واعمدة)Mass Assignment and JSON Columns(JSON

عند التعامل مع أعمدة JSON فيLaravel ، 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',
];
```

(السماح بتعيين الجماعي)Allowing Mass Assignment

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 = [];
```

(استثناءات التعيين الجماعي)Mass Assignment Exceptions

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.

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

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.

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']);
```

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

١. **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.

٢. **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؟

١. **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.**

(حذف النماذج)Deleting Models

```
use App\Models\Flight;
$flight = Flight::find(1);
$flight->delete();
```

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 (١) عند إدراج سجلات جديدة في الجدول).

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

(حذف جدول موجود من خلال مفتاحه الاساسي)Deleting an Existing Model by its Primary Key

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؟ الحذف الناعم(Soft Deletes)
Soft Deletingهي 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.

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

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.

(حذف النماذج باستخدام استعلام)Deleting Models Using Queries
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();
```

عند تنفيذ عملية **mass delete**باستخدامEloquent ، 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(الحذف المؤقت)
فيEloquent ، 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;
}
```

عند استخدام **SoftDeletes** فيEloquent ، 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.
إضافة عمود **deleted_at**إلى جدول قاعدة البيانات عند استخدام **Soft Deletes**فيLaravel ، 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();// إضافة
});
```

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

عند استدعاء الدالة **delete**على نموذج يستخدم **Soft Deletes**فيLaravel ، 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**

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

استعادة النماذج المحذوفة مؤقتاً (Restoring Soft Deleted Models)

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();// استعادة نموذج محذوف مؤقتاً
```

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(); // استعادة جماعية لجميع النماذج المحذوفة التي تطابق الشرط
```

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

```
$flight->history()->restore(); // استعادة النماذج المحذوفة في العلاقة
```

الحذف الدائم للنماذج(Permanently Deleting Models)

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

```
$flight->forceDelete(); // حذف النموذج بشكل دائم من قاعدة البيانات
```

الاستعلام عن النماذج المحذوفة مؤقتاً(Querying Soft Deleted Models)

تضمين النماذج المحذوفة مؤقتاً(Including Soft Deleted Models)

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(); // استرجاع جميع النماذج بما فيها المحذوفة
```

استرجاع النماذج المحذوفة فقط(Retrieving Only Soft Deleted Models)

الدالة onlyTrashed تسترجع فقط النماذج المحذوفة باستخدام الـ soft delete

```
$flights = Flight::onlyTrashed()
            ->where('airline_id', 1)
            ->get(); // استرجاع النماذج المحذوفة فقط
```

تقليم النماذج(Pruning Models)

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:

١. **Illuminate\Database\Eloquent\Prunable:** يقوم هذا **trait**بحذف النماذج التي لم تعد ضرورية بشكل فردي. يتم استدعاء الأحداث المرتبطة بالنموذج مثل **deleting**و .**deleted**

٢. **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**

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()); // السجلات الأقدم من شهر
   }
}
```

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
{
   // تنفيذ عمليات قبل الحذف الدائم
}
```

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(); // جدولة التنظيف اليومي
```

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(); // تنظيف نماذج محددة فقط
```

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(); // استثناء نماذج محددة من التنظيف
```

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 // اختبار عملية التنظيف بدون تنفيذ
```

عند استخدام **soft deleting** مع **Prunable** فيLaravel ، 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.

<u>التقليم الجماعي(Mass Pruning)</u>

عند استخدام **trait**الخاص بـ **Illuminate\Database\Eloquent\MassPrunable**مع النماذج فيLaravel ، 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:

١. **دالة** :**pruning**لن يتم استدعاؤها.
٢. **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()); // استعلام التقليم
   }
}
```

نسخ النماذج(Replicating Models)

فيLaravel ، 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' // تغيير نوع العنوان
]);
 $billing->save(); // حفظ النسخة الجديدة
```

لإستبعاد خاصية أو أكثر من عملية النسخ عند استخدام الدالة **replicate**فيLaravel ، 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', // استبعاد تاريخ آخر رحلة
   'last_pilot_id' // استبعاد معرف الطيار الأخير
]); إنشاء نسخة بدون الخصائص المستبعدة //
```

lecture 08 Eloquent – relationshipالملزمة الثانية فراس الدبعي

Eloquent: Relationships

مقدمة(Introduction)

غالبًا ما تكون جداول قواعد البيانات مرتبطة ببعضها البعض.

على سبيل المثال، قد تحتوي إحدى منشورات المدونة على العديد من التعليقات أو قد يكون الطلب مرتبطًا بالمستخدم الذي قدمه.

يجعل Eloquent إدارة هذه العلاقات والعمل بها أمرًا سهلاً، ويدعم مجموعة متنوعة من العلاقات الشائعة:

تحديد العلاقات(Defining Relationships)

نظرًا لأن العلاقات تعمل أيضًا كمنشئات قوية للاستعلامات، فإن تعريف العلاقات كدوال يوفر قدرات قوية لتسلسل استدعاءات الدوال والاستعلام.

```
$user->posts()->where('active', 1)->get();
```

Has One (One to One / Has One) / واحد لواحد

العلاقة من واحد إلى واحد هي نوع أساسي من علاقات قاعدة البيانات.

على سبيل المثال، قد يرتبط model المستخدم بـ model هاتف واحد.

لتحديد هذه العلاقة، سنضع الدالة phone على model المستخدم.

يجب أن تستدعي الدالة phone الدالة hasOne وتعيد نتيجتها.

تتوفر الدالة hasOne لنموذجك عبر فئة القاعدة Illuminate\Database\Eloquent\Model للنموذج:

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

class User extends Model
{
   /**
    * الحصول على الهاتف المرتبط بالمستخدم.
    * Get the phone associated with the user.
    */
   public function phone(): HasOne        // تعريف علاقة واحد لواحد: للمستخدم هاتف واحد (ينتمي إليه).
   {
      return $this->hasOne(Phone::class);       // المعامل: اسم موديل الهاتف المرتبط (Phone::class)
   }
}
```

المعامل الأول الذي يتم تمريره إلى الدالة hasOne هي اسم فئة الـ model ذات الصلة.

بمجرد تحديد العلاقة، يمكننا استرداد السجل ذي الصلة باستخدام خصائص Eloquent الديناميكية.

تتيح لك الخصائص الديناميكية الوصول إلى دوال العلاقة كما لو كانت خصائص محددة في النموذج:

```
$phone = User::find(1)->phone; // جلب الهاتف المرتبط بالمستخدم الذي معرفه ١ باستخدام الخاصية الديناميكية (ليست استدعاء دالة).
```

يحدد Eloquent المفتاح الاجنبي للعلاقة استنادًا إلى اسم النموذج الرئيسي.

في هذه الحالة، يُفترض تلقائيًا أن نموذج الهاتف يحتوي على مفتاح اجنبي user_id.

إذا كنت ترغب في تجاوز هذه الطريقة لتعريف المفتاح الاجنبي، فيمكنك تمرير معامل ثاني إلى دالة hasOne:

```
return $this->hasOne(Phone::class, 'foreign_key');
```

بالإضافة إلى ذلك، يفترض Eloquent أن المفتاح الاجنبي يجب أن يحتوي على قيمة تطابق عمود المفتاح الأساسي للمفتاح الرئيسي. بعبارة أخرى، سيبحث Eloquent عن قيمة عمود معرف المستخدم في عمود user_id لسجل الهاتف.

إذا كنت ترغب في أن تستخدم العلاقة قيمة مفتاح أساسي غير المعرف أو خاصية $primaryKey الخاصة بالـ model، فيمكنك تمرير وسيط ثالث إلى الدالة hasOne:

```
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

تحديد عكس العلاقة(Defining the Inverse of the Relationship)

لذلك، يمكننا الوصول إلى model الهاتف من model المستخدم الخاص بنا.

بعد ذلك، دعنا نحدد علاقة على model الهاتف تسمح لنا بالوصول إلى المستخدم الذي يمتلك الهاتف.

يمكننا تحديد عكس علاقة hasOne باستخدام الدالة belongTo:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
class Phone extends Model
{
   /**
    * الحصول على المستخدم الذي يمتلك الهاتف.
    * Get the user that owns the phone.
```

```
    */
    public function user(): BelongsTo       // تعريف عكس العلاقة: الهاتف ينتمي إلى مستخدم واحد.
    {
        return $this->belongsTo(User::class);       // المعامل: اسم موديل المستخدم المالك (User::class)
    }
}
```

عند استدعاء الدالة user ، سيحاول Eloquent العثور على model مستخدم يحتوي على معرف يتطابق مع عمود user_id في model الهاتف.
يقوم Eloquent بتحديد اسم المفتاح الاجنبي من خلال فحص اسم دالة العلاقة وإضافة اللاحقة id_ إلى اسم الدالة. لذا، في هذه الحالة، يفترض Eloquent أن model الهاتف يحتوي على عمود user_id.
ومع ذلك، إذا لم يكن المفتاح الاجنبي في نموذج الهاتف هو user_id، فيمكنك تمرير اسم مفتاح مخصص كمعامل ثان الى الدالة belongTo:

```
/**
* Get the user that owns the phone. * الحصول على المستخدم الذي يمتلك الهاتف.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key');
}
```

إذا كان model الرئيسي لا يستخدم id كمفتاح أساسي، أو كنت ترغب في العثور على النموذج المرتبط باستخدام عمود مختلف، فيمكنك تمرير وسيط ثالث إلى الدالة belongTo لتحديد المفتاح المخصص للجدول الرئيسي:

```
/**
 * Get the user that owns the phone.
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

Has Many (One to Many / Has Many) / واحد لكثير

تُستخدم علاقة واحد إلى كثير لتحديد العلاقات حيث يكون model واحد هو الأصل لـ model فرعي واحد أو أكثر. على سبيل المثال، قد تحتوي مشاركة المدونة على عدد لا نهائي من التعليقات.
مثل جميع علاقات Eloquent الأخرى، يتم تحديد علاقات واحد إلى كثير من خلال تحديد دالة على نموذج Eloquent الخاص بك:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;
class Post extends Model
{
    /**
     * Get the comments for the blog post.     * الحصول على التعليقات الخاصة بمنشور المدونة.
     */
    public function comments(): HasMany       // تعريف علاقة واحد لكثير: للمنشور العديد من التعليقات.
    {
        return $this->hasMany(Comment::class);       // المعامل: اسم موديل التعليق المرتبط (Comment::class)

    }
}
```

تذكر أن Eloquent سيحدد تلقائيًا عمود المفتاح الاجنبي المناسب لـ model التعليق.
وفقًا للطريقة التي يستخدمها الـ laravel، سيأخذ Eloquent اسم "snake case" للـ model الأساسي ويضيف إليه اللاحقة id_.
لذا، في هذا المثال، سيفترض Eloquent أن عمود المفتاح الاجنبي في model التعليق هو post_id.
بمجرد تعريف دالة العلاقة، يمكننا الوصول إلى مجموعة التعليقات ذات الصلة من خلال الوصول إلى خاصية التعليقات.
تذكر، بما أن Eloquent يوفر "خصائص علاقة ديناميكية"، فيمكننا الوصول إلى دوال العلاقة كما لو كانت محددة كخصائص في الـ model :

```
use App\Models\Post;
$comments = Post::find(1)->comments;// استخدام الخاصية الديناميكية comments (ليست استدعاء دالة)
foreach ($comments as $comment) {
    // ...
}
```

نظرًا لأن جميع العلاقات تعمل أيضًا كمنشئين للاستعلامات، فيمكنك إضافة قيود إضافية إلى استعلام العلاقة عن طريق استدعاء دالة التعليقات ومواصلة ربط الشروط بالاستعلام:

```
$comment = Post::find(1)->comments()
                ->where('title', 'foo')
                ->first();// استدعاء علاقة comments() كدالة لبناء استعلام، ثم إضافة شرط (where) للعنوان، ثم جلب أول نتيجة.
```

مثل دالة hasOne، يمكنك أيضًا تجاوز المفاتيح الأجنبية والمحلية عن طريق تمرير وسائط إضافية إلى الدالة hasMany:

```
return $this->hasMany(Comment::class, 'foreign_key'); // تحديد المفتاح الخارجي
return $this->hasMany(Comment::class, 'foreign_key', 'local_key'); // تحديد المفتاح الخارجي والمفتاح المحلي
```

الترطيب التلقائي للنماذج الأصلية على النماذج الفرعية(Automatically Hydrating Parent Models on Children)

عند استخدام التحميل السريع لـ Eloquent، قد تنشأ مشكلات استعلام "N + 1" إذا حاولت الوصول إلى الـ model الرئيسي من model فرعي أثناء التنقل عبر الـ models الفرعية:

```
$posts = Post::with('comments')->get(); // التحميل السريع (Eager Loading) للتعليقات مع المنشورات
foreach ($posts as $post) {
   foreach ($post->comments as $comment) {
      echo $comment->post->title; // هنا تحدث مشكلة !N+1 يتم استعلام منفصل لكل تعليق لجلب المنشور الخاص به.
   }
}
```

في المثال أعلاه، ظهرت مشكلة استعلام "N + 1" لأنه، على الرغم من تحميل التعليقات بسرعة لكل Post model، فإن Eloquent لا يقوم تلقائيًا بربط Post الرئيسي في كل Comment model فرعي.

إذا كنت ترغب في أن يقوم Eloquent تلقائيًا بعمل hydrate للـ parent models على فروعها، فيمكنك استدعاء الدالة chaperone عند تعريف علاقة hasMany:

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

class Post extends Model
{
   /**
    * Get the comments for the blog post. * الحصول على التعليقات الخاصة بمنشور المدونة.
    */
   public function comments(): HasMany
   {       // استخدام chaperone() لتجنب مشاكل N+1 عند الوصول للنموذج الأصلي من النموذج الفرعي
      return $this->hasMany(Comment::class)->chaperone();
   }
}
```

تتجنب الدالة **chaperone()** الاستعلامات غير المتوقعة N+1 من خلال الارتباط مرة أخرى بالعنصر الرئيسي بعد تشغيل استعلام العلاقة. ستربط العلاقة أعلاه نموذج Post المناسب في العلاقة الصحيحة في حالات التعليق (مع بقاء النطاقات سليمة).

أو، إذا كنت ترغب في الاشتراك في الربط التلقائي للوالد في وقت التنفيذ، فيمكنك استدعاء الـ chaperone model عند تحميل العلاقة:

```
use App\Models\Post;
 $posts = Post::with([
   'comments' => fn ($comments) => $comments->chaperone(), // تطبيق chaperone عند التحميل السريع
])->get();
```

Belongs To (One to Many (Inverse) / Belongs To) / واحد لكثير (عكس)

الآن بعد أن أصبح بإمكاننا الوصول إلى كافة تعليقات المنشور، دعنا نحدد علاقة للسماح للتعليق بالوصول إلى المنشور الأصلي الخاص به.

لتحديد عكس علاقة hasMany، قم بتعريف دالة علاقة على النموذج الفرعي الذي يستدعي الدالة belongTo:

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

class Comment extends Model
{
   /**
    * Get the post that owns the comment.   * الحصول على المنشور الذي يمتلك التعليق.
```

```
    */
    public function post(): BelongsTo    // تعريف عكس علاقة واحد لكثير: التعليق ينتمي إلى منشور واحد.
    {
        return $this->belongsTo(Post::class);    // المعامل: اسم موديل المنشور المالك (Post::class)
    }
}
```

بمجرد تحديد العلاقة، يمكننا استرداد المنشور الأصلي للتعليق من خلال الوصول إلى "خاصية العلاقة الديناميكية" في المنشور:

```
use App\Models\Comment;
$comment = Comment::find(1);
return $comment->post->title; // الوصول إلى المنشور الأصلي عبر الخاصية الديناميكية post
```

في المثال أعلاه، سيحاول Eloquent العثور على Post model يحتوي على id يتطابق مع عمود post_id في نموذج التعليق.
يقوم Eloquent بتحديد اسم المفتاح الاجنبي الافتراضي من خلال فحص اسم دالة العلاقة وإضافة _id إلى اسم الدالة متبوعًا باسم عمود المفتاح الأساسي للـ model الرئيسي. لذا، في هذا المثال، سيفترض Eloquent أن المفتاح الاجنبي لـ model الـ Post في جدول التعليقات هو post_id.
إذا كان المفتاح الأجنبي لا يتبع الطريقة التلقائية لتسميته، يمكنك تمرير المفتاح الأجنبي الخاص بالعلاقة كمعامل ثان للدالة belonTo() كما يلي:

```
/**
 * Get the post that owns the comment.    * الحصول على المنشور الذي يمتلك التعليق.
 */
public function post(): BelongsTo
{
    return $this->belongsTo(Post::class, 'foreign_key');        // تحديد اسم المفتاح الخارجي في جدول التعليقات يدوياً إذا لم يكن 'post_id'
}
```

إذا كان الـ model الرئيسي لا يستخدم الـ id كمفتاح رئيسي، او تم استخدام عمود اخر لربط الجدولين، تستطيع تمرير هذا المفتاح الموجود في الـ model الرئيسي كمعامل ثالث الى الدالة belonTo() كما يلي:

```
/**
 * Get the post that owns the comment.    * الحصول على المنشور الذي يمتلك التعليق.
 */
public function post(): BelongsTo
{// المعامل الثالث: اسم المفتاح المحلي (owner key) في جدول المنشورات (إذا لم يكن 'id')
    return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

النماذج الافتراضية **(Default Models)**

تسمح لك العلاقات belongTo و hasOne و hasOneThrough و morphOne بتحديد model افتراضي سيتم إرجاعه إذا كانت العلاقة المحددة فارغة.
يُشار إلى هذا النمط غالبًا باسم نمط Null Object ويمكن أن يساعد في إزالة عمليات التحقق الشرطية في الكود الخاص بك.

```
/**
 * Get the author of the post.    * الحصول على مؤلف المنشور.
 */
public function user(): BelongsTo
{        // withDefault() ترجع نموذج User افتراضي إذا لم يكن للمنشور مؤلف
    return $this->belongsTo(User::class)->withDefault();
}
```

لملء النموذج الافتراضي بالسمات، يمكنك تمرير مصفوفة أو عبارة closure إلى الدالة withDefault:

```
/**
 * Get the author of the post. * الحصول على مؤلف المنشور.
 */
public function user(): BelongsTo
{// مع withDefault الافتراضي يمكنك تحديد بيانات للنموذج (مثل الاسم 'Guest Author')
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Guest Author',
    ]);
}

/**
 * Get the author of the post.    * الحصول على مؤلف المنشور.
 */
public function user(): BelongsTo
```

```
{// لتعيين بيانات النموذج الافتراضي بشكل ديناميكي Closure أو استخدام
   return $this->belongsTo(User::class)->withDefault(function (User $user, Post $post) {
      $user->name = 'Guest Author';
   });
}
```

الاستعلام عن علاقات Belongs To (Querying Belongs To Relationships)

عند الاستعلام عن أبناء علاقة "belongs to "، يمكنك إنشاء شرط where يدويًا لاسترداد Eloquent models التي ينطبق عليها الشرط:

```
use App\Models\Post;
$posts = Post::where('user_id', $user->id)->get();// استعلام يدوي للمنشورات التي user_id يساوي معرف مستخدم معين
```

ومع ذلك، قد تجد أنه من الأكثر ملاءمة استخدام الدالة whereBelongsTo، والتي ستحدد تلقائيًا العلاقة المناسبة والمفتاح الاجنبي للنموذج المحدد:

```
$posts = Post::whereBelongsTo($user)->get();// استعلام تلقائي عن المنشورات المرتبطة بمستخدم معين ($user)
```

يمكنك أيضًا توفير instance collection للدالة whereBelongsTo.

وعند القيام بذلك، سيسترد Laravel الـ models التي تنتمي إلى أي من parent models ضمن المجموعة:

```
$users = User::where('vip', true)->get();
$posts = Post::whereBelongsTo($users)->get();
```

بشكل افتراضي، سيحدد Laravel العلاقة المرتبطة بالـ model المحدد استنادًا إلى اسم فئة الـmodel ؛ ومع ذلك، يمكنك تحديد اسم العلاقة يدويًا عن طريق تمريره كمعامل ثان للدالة whereBelongsTo:

```
$posts = Post::whereBelongsTo($user, 'author')->get(); // تحديد اسم العلاقة يدوياً إذا لم يكن 'user' ('author' مثلاً)
```

Has One of Many

في بعض الأحيان قد يكون للـ model العديد من الـ models ذات الصلة، ولكنك تريد استرداد "الأحدث" أو "الأقدم" من النماذج ذات الصلة بالعلاقة بسهولة.

على سبيل المثال، قد يكون model المستخدم مرتبطا بالعديد من models الطلبات، ولكنك تريد تحديد طريقة ملائمة للتفاعل مع الطلب الأحدث الذي قدمه المستخدم.

يمكنك إنجاز ذلك باستخدام نوع العلاقة hasOne المدمج مع دوال الـ ofMany:

```
/**
 * Get the user's most recent order. * الحصول على أحدث طلب للمستخدم.
 */
public function latestOrder(): HasOne
{
   return $this->hasOne(Order::class)->latestOfMany();
}
```

بالمثل يمكنك تحديد دوال تسترجع الاقدم، او الأول، من الـ model المرتبط بالعلاقة.

```
/**
 * Get the user's oldest order.    * الحصول على أقدم طلب للمستخدم.
 */
public function oldestOrder(): HasOne
{
   return $this->hasOne(Order::class)->oldestOfMany();
}
```

بشكل افتراضي، ستقوم الدالتين latestOfMany وoldestOfMany باسترداد أحدث أو أقدم model مرتبط استنادًا إلى المفتاح الأساسي للـ model ، والذي يجب أن يكون قابلاً للفرز.

ومع ذلك، قد ترغب أحيانًا في استرداد model واحد من علاقة اوسع باستخدام معايير فرز مختلفة.

على سبيل المثال، باستخدام الدالة ofMany ، يمكنك استرداد الطلب الأكثر تكلفة للمستخدم.

تقبل الدالة ofMany العمود القابل للفرز كمعامل أول، كما تقبل دالة التجميع (الحد الأدنى أو الأقصى) التي سيتم تطبيقها عند الاستعلام عن النموذج المرتبط:

```
/**
 * Get the user's largest order.    * الحصول على أكبر طلب للمستخدم.
 */
public function largestOrder(): HasOne
{
   return $this->hasOne(Order::class)->ofMany('price', 'max');
}
```

تحويل علاقات "Many" إلى علاقات Has One (Converting "Many" Relationships to Has One Relationships)

في كثير من الأحيان، عند استرداد model واحد باستخدام الدوال latestOfMany، أو olderOfMany، أو ofMany، يكون لديك بالفعل علاقة "has many" محددة لنفس الـ model .

بطريقة أفضل، يسمح لك Laravel بتحويل هذه العلاقة بسهولة إلى علاقة "has one" من خلال استدعاء الدالة one في العلاقة:

```
/**
 * Get the user's orders.     * الحصول على طلبات المستخدم.
 */
public function orders(): HasMany
{
   return $this->hasMany(Order::class);
}

/**
* Get the user's largest order.     * الحصول على أكبر طلب للمستخدم.
 */
public function largestOrder(): HasOne
{
   return $this->orders()->one()->ofMany('price', 'max');
}
```

علاقات Has One of Many المتقدمة(Advanced Has One of Many Relationships)

من الممكن إنشاء علاقات "has one of many" أكثر تقدمًا. على سبيل المثال، قد يحتوي model المنتج على العديد من نماذج الأسعار المرتبطة به والتي يتم الاحتفاظ بها في النظام حتى بعد نشر التسعير الجديد.
بالإضافة إلى ذلك، قد يكون من الممكن نشر بيانات التسعير الجديدة للمنتج مسبقًا لكي تسري في تاريخ مستقبلي عبر عمود published_at.
لذلك، باختصار، نحن بحاجة إلى استرداد أحدث الأسعار المنشورة حيث لا يكون تاريخ النشر في المستقبل.
بالإضافة إلى ذلك، إذا كان لسعرين نفس تاريخ النشر، فسوف نفضل السعر الذي يحمل أكبر معرف.
لإنجاز ذلك، يجب علينا تمرير مصفوفة إلى الدالة ofMany التي تحتوي على الأعمدة القابلة للفرز والتي تحدد أحدث سعر.
بالإضافة إلى ذلك، سيتم تمرير عبارة closure كمعامل ثاني للدالة ofMany.
سيكون هذا الـ closure مسؤولاً عن إضافة قيود إضافية لتاريخ نشر استعلام العلاقة:

```
/**
 * Get the current pricing for the product.     * الحصول على التسعير الحالي للمنتج.
 */
public function currentPricing(): HasOne
{
   return $this->hasOne(Price::class)->ofMany([
      'published_at' => 'max', // نريد أحدث تاريخ نشر
      'id' => 'max', // نريد أكبر published_at إذا تساوى ، id
   ], function (Builder $query) {
      $query->where('published_at', '<', now());// أن يكون published_at قيد إضافي: في الماضي (ليس مستقبلي)
   });
}
```

Has One Through

الـ "has-one-through" علاقة واحد لواحد لـ model مع model آخر.
ومع ذلك، تشير هذه العلاقة إلى أنه يمكن يرتبط الـ model الحالي مع كائن model آخر عبر علاقته مع model ثالث.
على سبيل المثال، في تطبيق ورشة إصلاح المركبات، قد يتم ربط كل model ميكانيكي بـ model سيارة واحد، وقد يتم ربط كل model سيارة بـ model مالك واحد. وبعلاقة غير مباشرة قد يرتبط model ميكانيكي بـ model المالك عبر model السيارة.
على الرغم من عدم وجود علاقة مباشرة بين الميكانيكي والمالك داخل قاعدة البيانات، إلا أن الميكانيكي يستطيع الوصول إلى المالك من خلال modelالسيارة.
دعنا نلقي نظرة على الجداول اللازمة لتحديد هذه العلاقة:

```
mechanics
   id - integer
   name - string

cars
   id - integer
   model - string
   mechanic_id - integer

owners
   id - integer
   name - string
```

```
car_id - integer
```

الآن بعد أن قمنا بفحص بنية الجدول للعلاقة، دعنا نحدد العلاقة في model الميكانيكي:

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

class Mechanic extends Model
{
   /**
    * Get the car's owner.     * الحصول على مالك السيارة.
    */
   public function carOwner(): HasOneThrough
   {       // المعامل ١: النموذج النهائي (Owner)       // النموذج الوسيط ٢: المعامل (Car)
      return $this->hasOneThrough(Owner::class, Car::class);
   }
}
```

المعامل الاول التي تم تمريره إلى الدالة hasOneThrough هي اسم model النهائي الذي نرغب الوصول إليه، بينما المعامل الثاني هو اسم model الوسيط.

أو، إذا تم تعريف العلاقات ذات الصلة بالفعل على جميع الmodels المشاركة في العلاقة، فيمكنك تعريف علاقة "has-one-through" بطلاقة من خلال استدعاء الدالة through وتوفير أسماء تلك العلاقات.

على سبيل المثال، إذا كان model الميكانيكي يحتوي على علاقة سيارات وكان model السيارة يحتوي على علاقة مالك، فيمكنك تعريف علاقة "من طرف واحد" تربط الميكانيكي بالمالك على النحو التالي:

```
// String based syntax...
return $this->through('cars')->has('owner');

// Dynamic syntax...
return $this->throughCars()->hasOwner();
```

اتفاقيات المفاتيح(Key Conventions)

سيتم استخدام طريقة المفاتيح الاجنبية النموذجية المقدمة من الـ Eloquent عند تنفيذ استعلامات العلاقة.

إذا كنت ترغب في تخصيص مفاتيح العلاقة، فيمكنك تمريرها كمعامل ثالث ومعامل رابع إلى الدالة hasOneThrough.

المعامل الثالث هو اسم المفتاح الاجنبي في model الوسيط.

المعامل الرابع هو اسم المفتاح الاجنبي في model النهائي.

المعامل الخامس هي المفتاح المحلي، بينما المعامل السادس هو المفتاح المحلي لل model الوسيط:

```
class Mechanic extends Model
{
   /**
    * Get the car's owner.     * الحصول على مالك السيارة.
    */
   public function carOwner(): HasOneThrough
   {
      return $this->hasOneThrough(
         Owner::class, ,    // النموذج النهائي
         Car::class,      // النموذج الوسيط
         'mechanic_id',   // المفتاح الخارجي على جدول السيارات (الوسيط) الذي يشير إلى الميكانيكي
         'car_id',        // المفتاح الخارجي على جدول المالكين (النهائي) الذي يشير إلى السيارة
         'id',            // المفتاح المحلي على جدول الميكانيكيين
         'id'             // المفتاح المحلي على جدول السيارات
      );
   }
}
```

أو، كما تمت مناقشته سابقًا، إذا تم بالفعل تعريف العلاقات ذات الصلة على جميع الـmodels المشاركة في العلاقة، فيمكنك تعريف علاقة "has-one-through" بطلاقة عن طريق استدعاء الدالة through وتمرير أسماء تلك العلاقات.

Has Many Through

توفر علاقة "has-many-through" طريقة ملائمة للوصول إلى العلاقات البعيدة عبر علاقة وسيطة. على سبيل المثال، لنفترض أننا نبني منصة نشر مثل Laravel Vapor.
قد يتمكن model المشروع من الوصول إلى العديد من models النشر من خلال model بيئة وسيطة.
باستخدام هذا المثال، يمكنك بسهولة جمع كل عمليات النشر لمشروع معين.
دعنا نلقي نظرة على الجداول المطلوبة لتحديد هذه العلاقة:

```
projects
    id - integer
    name - string

environments
    id - integer
    project_id - integer
    name - string

deployments
    id - integer
    environment_id - integer
    commit_hash - string
```

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

class Project extends Model
{
    /**
     * Get all of the deployments for the project.     * الحصول على كافة عمليات النشر للمشروع.
     */
    public function deployments(): HasManyThrough
    {        // المعامل ١: النموذج النهائي (Deployment)        // المعامل ٢: النموذج الوسيط (Environment)
        return $this->hasManyThrough(Deployment::class, Environment::class);
    }
}
```

المعامل الأول الذي تم تمريره إلى الدالة hasManyThrough هو اسم الـ model النهائي الذي نرغب في الوصول إليه، بينما المعامل الثاني هو اسم الـ model الوسيط.
أو، إذا تم تعريف العلاقات ذات الصلة بالفعل على جميع الـ models المشاركة في العلاقة، فيمكنك تعريف علاقة "has-many-through" بطلاقة عن طريق استدعاء الدالة through وتوفير أسماء تلك العلاقات.
على سبيل المثال، إذا كان model المشروع يحتوي على علاقة بيئات وكان نموذج البيئة يحتوي على علاقة عمليات نشر، فيمكنك تعريف علاقة "has-many-through" التي تربط المشروع وعمليات النشر على النحو التالي:

```
// String based syntax...
return $this->through('environments')->has('deployments');

// Dynamic syntax...
return $this->throughEnvironments()->hasDeployments();
```

على الرغم من أن جدول model النشر لا يحتوي على عمود project_id، فإن علاقة hasManyThrough توفر الوصول إلى عمليات نشر المشروع عبر $project->deployments.
لاسترجاع هذه الـ models، يقوم Eloquent بفحص عمود project_id في جدول model البيئة الوسيط. بعد العثور على معرفات البيئة ذات الصلة، يتم استخدامها لاستعلام جدول model النشر.

اتفاقيات المفاتيح(Key Conventions)

سيتم استخدام تسميات المفاتيح الاجنبية النموذجية لـ Eloquent عند تنفيذ استعلامات العلاقة.
إذا كنت ترغب في تخصيص مفاتيح العلاقة، فيمكنك تمريرها كمعاملين ثالث ورابع إلى الدالة hasManyThrough.
المعامل الثالث هو اسم المفتاح الاجنبي في الـ model الوسيط.
المعامل الرابع هو اسم المفتاح الاجنبي في الـ model النهائي.

المعامل الخامس هو المفتاح المحلي، بينما المعامل السادس هي المفتاح المحلي لل model الوسيط:

```
class Project extends Model
{
   public function deployments(): HasManyThrough
   {
      return $this->hasManyThrough(
         Deployment::class,     // النموذج النهائي
         Environment::class,    // النموذج الوسيط
         'project_id',          // المفتاح الخارجي على جدول البيئات (الوسيط) الذي يشير إلى المشروع
         'environment_id',      // المفتاح الخارجي على جدول النشر (النهائي) الذي يشير إلى البيئة
         'id',                  // المفتاح المحلي على جدول المشاريع
         'id'                   // المفتاح المحلي على جدول البيئات
      );
   }
}
```

أو، كما تمت مناقشته سابقًا، إذا تم بالفعل تعريف العلاقات ذات الصلة على جميع الـ models المشاركة في العلاقة، فيمكنك تعريف علاقة "has-many-through" بطلاقة عن طريق استدعاء طريقة through وتوفير أسماء تلك العلاقات.

يقدم هذا النهج ميزة إعادة استخدام التسميات الرئيسية المحددة بالفعل في العلاقات الموجودة:

```
// String based syntax...
return $this->through('environments')->has('deployments');

// Dynamic syntax...
return $this->throughEnvironments()->hasDeployments();
```

علاقات متعددة(Many to Many Relationships)

العلاقات متعدد الى متعدد تكون أكثر تعقيدًا قليلًا من العلاقات hasOne وhasMany.

ومن الأمثلة على العلاقات متعدد الى متعدد المستخدم الذي لديه العديد من الـ roles ويتم مشاركة هذه الـ roles أيضًا من قبل مستخدمين آخرين في التطبيق.

على سبيل المثال، قد يتم تعيين مستخدم لوظيفة "Author" و"Editor"؛ ومع ذلك، قد يتم تعيين هذه الوظائف لمستخدمين آخرين أيضًا. وبالتالي، فإن المستخدم لديه العديد من الـ roles والـrole لديه العديد من المستخدمين.

هيكل الجدول(Table Structure)

لتمثيل هذه العلاقة، هناك حاجة إلى ثلاثة جداول قاعدة بيانات: المستخدمون والأدوار ومستخدم الدور.

يتم اشتقاق جدول مستخدم الدور من الترتيب الأبجدي لأسماء النماذج ذات الصلة ويحتوي على عمودي معرف المستخدم ومعرف الدور. يتم استخدام هذا الجدول كجدول وسيط يربط بين المستخدمين والأدوار.

تذكر، بما أن الدور يمكن أن ينتمي إلى العديد من المستخدمين، فلا يمكننا ببساطة وضع عمود user_id في جدول الأدوار. وهذا يعني أن الدور يمكن أن ينتمي إلى مستخدم واحد فقط.

من أجل توفير الدعم لتخصيص الأدوار لمستخدمين متعددين، يلزم وجود جدول role_user.

يمكننا تلخيص بنية جدول العلاقة على النحو التالي:

```
users
   id - integer
   name - string

roles
   id - integer
   name - string

role_user
   user_id - integer
   role_id - integer
```

هيكل النموذج(Model Structure)

يتم تعريف العلاقات العديد إلى العديد من خلال كتابة دالة تعيد نتيجة الدالة belongToMany.

يتم توفير الدالة belongToMany بواسطة الفئة الأساسية Illuminate\Database\Eloquent\Model التي تستخدمها جميع نماذج Eloquent models في تطبيقك.

على سبيل المثال، دعنا نعرف الدالة roles في model المستخدم الخاص بنا.

المعامل الأول التي يتم تمريره إلى هذه الدالة هي اسم فئة الـ model ذات الصلة:

```
<?php
```

```
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class User extends Model
{
   /**
    * The roles that belong to the user.     * الأدوار التي ينتمي إليها المستخدم.
    */
   public function roles(): BelongsToMany
   {       // تعريف علاقة متعددة: المستخدم ينتمي إلى العديد من الأدوار
      return $this->belongsToMany(Role::class);       // المعامل: اسم موديل الدور المرتبط (Role::class)
   }
}
```

بمجرد تحديد العلاقة، يمكنك الوصول إلى أدوار المستخدم باستخدام خاصية العلاقة الديناميكية للأدوار:

```
use App\Models\User;
 $user = User::find(1);
 foreach ($user->roles as $role) {
   // ...
}
```

نظرًا لأن جميع العلاقات تعمل أيضًا كمنشئين للاستعلامات، فيمكنك إضافة قيود إضافية إلى استعلام العلاقة عن طريق استدعاء دالة roles ومواصلة ربط الشروط بالاستعلام:

```
$roles = User::find(1)->roles()->orderBy('name')->get();
```

لتحديد اسم الجدول الوسيط للعلاقة، سيقوم Eloquent بربط اسمي الـ models المرتبطين بالترتيب الأبجدي. ومع ذلك، يمكنك تجاوز هذه التسمية.
يمكنك القيام بذلك عن طريق تمرير معامل ثاني إلى الدالة belongToMany:

```
return $this->belongsToMany(Role::class, 'role_user'); // تحديد اسم جدول الربط الوسيط يدوياً
```

بالإضافة إلى تخصيص اسم الجدول الوسيط، يمكنك أيضًا تخصيص أسماء الأعمدة الخاصة بالمفاتيح الموجودة في الجدول عن طريق تمرير معاملات إضافية إلى الدالة belongToMany.
المعامل الثالثة هي اسم المفتاح الاجنبي للنموذج الذي تقوم بتعريف العلاقة عليه، بينما المعامل الرابعة هي اسم المفتاح الاجنبي للنموذج الذي تقوم بالانضمام إليه:

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

تحديد عكس العلاقة(Defining the Inverse of the Relationship)

لتحديد "العكس" لعلاقة متعدد الى متعدد، يجب عليك تحديد دالة على الـ model ذي الصلة والتي تقوم أيضًا بإرجاع نتيجة الدالة belongToMany.
لإكمال مثال المستخدم/الدور، دعنا نحدد الدالة users() في نموذج الدور:

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

class Role extends Model
{
   /**
    * The users that belong to the role.     * المستخدمون الذين ينتمون إلى الدور.
    */
   public function users(): BelongsToMany
   {
      return $this->belongsToMany(User::class);
   }
}
```

كما ترى، يتم تعريف العلاقة بنفس الطريقة تمامًا مثل نظيرها في model المستخدم باستثناء الإشارة إلى نموذج App\Models\User. نظرًا لأننا نعيد استخدام الدالة belongToMany، فإن جميع خيارات تخصيص الجدول والمفاتيح المعتادة متاحة عند تعريف "العكس" للعلاقات من العديد إلى العديد.

استرداد بيانات جدول الوسيط(Retrieving Intermediate Table Columns)

كما تعلمت بالفعل، يتطلب العمل مع العلاقات المتعدد إلى المتعدد وجود جدول وسيط. يوفر Eloquent بعض الطرق المفيدة جدًا للتفاعل مع هذا الجدول.
على سبيل المثال، لنفترض أن modelالمستخدم الخاص بنا يحتوي على العديد من models الأدوار التي يرتبط بها.
بعد الوصول إلى هذه العلاقة، يمكننا الوصول إلى الجدول الوسيط باستخدام الـ pivot attribute في الـ model:

```
use App\Models\User;

$user = User::find(1);
foreach ($user->roles as $role) {
    echo $role->pivot->created_at; // الوصول إلى بيانات إضافية في جدول الربط (pivot) مثل created_at
}
```

لاحظ أن كل model دور نسترده يتم تعيين سمة pivot له تلقائيًا.
تحتوي هذه السمة على model يمثل الجدول الوسيط.
بشكل افتراضي، ستكون مفاتيح الـ model فقط موجودة في pivot model.
إذا كان الجدول الوسيط الخاص بك يحتوي على سمات إضافية، فيجب عليك تحديدها عند تعريف العلاقة:

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by'); // تضمين أعمدة إضافية من جدول الربط
```

إذا كنت ترغب في أن يحتوي الجدول الوسيط لديك على طوابع زمنية created_at و updated_at يتم صيانتها تلقائيًا بواسطة Eloquent، استدعي الدالة withTimestamps عند تعريف العلاقة:

```
return $this->belongsToMany(Role::class)->withTimestamps(); // تضمين الطوابع الزمنية في جدول الربط
```

يجب أن تحتوي الجداول الوسيطة التي تستخدم الطوابع الزمنية التي يتم صيانتها تلقائيًا في Eloquent على عمودي الطابع الزمني created_at وupdate_at.

تخصيص سمة Pivot (Customizing the Pivot Attribute Name)

كما ذكرنا سابقًا، يمكن الوصول إلى السمات من الجدول الوسيط في الـ models عبر سمة pivot.
ومع ذلك، يمكنك تخصيص اسم هذه السمة لتعكس غرضها بشكل أفضل داخل تطبيقك.
على سبيل المثال، إذا كان تطبيقك يحتوي على مستخدمين قد يشتركون في البث الصوتي، فمن المحتمل أن يكون لديك علاقة متعددة إلى متعددة بين المستخدمين والبث الصوتي.
إذا كانت هذه هي الحالة، فقد ترغب في إعادة تسمية سمة الجدول الوسيط لديك إلى subscription بدلاً من pivot.
ويمكن القيام بذلك باستخدام الدالة as عند تعريف العلاقة:

```
return $this->belongsToMany(Plan::class)
    ->as('subscription') // تخصيص اسم سمة الجدول الوسيط لتكون 'subscription' بدلاً من 'pivot'
    ->withTimestamps();
```

بمجرد تحديد سمة الجدول الوسيط المخصصة، يمكنك الوصول إلى بيانات الجدول الوسيط باستخدام الاسم المخصص:

```
$users = User::with('plan')->get();
foreach ($users as $user) {
    echo $user->subscription->created_at; // الوصول إلى البيانات عبر السمة المخصصة 'subscription'
}
```

تصفية الاستعلامات عبر أعمدة الجدول الوسيط(Filtering Queries via Intermediate Table Columns)

يمكنك أيضًا تصفية النتائج التي تم إرجاعها بواسطة استعلامات علاقة belongToMany باستخدام الدوال wherePivot، وwherePivotIn، وwherePivotNotIn، وwherePivotBetween، وwherePivotNotBetween، وwherePivotNull، وwherePivotNotNull عند تعريف العلاقة:

```
return $this->belongsToMany(Role::class)
    ->wherePivot('approved', true); // إضافة شرط على عمود 'approved' في جدول الربط

return $this->belongsToMany(Role::class)
    ->wherePivotIn('priority', [1, 2]); // إضافة شرط whereIn على عمود 'priority'

return $this->belongsToMany(Role::class)
    ->wherePivotNotIn('priority', [1, 2]); // إضافة شرط whereNotIn على عمود 'priority'

return $this->belongsToMany(Bonus::class)
    ->wherePivotBetween('created_at', ['2023-01-01 00:00:00', '2023-12-31 23:59:59']); // إضافة شرط whereBetween

return $this->belongsToMany(Bonus::class)
    ->wherePivotNotBetween('created_at', ['2023-01-01 00:00:00', '2023-12-31 23:59:59']); // إضافة شرط
whereNotBetween

return $this->belongsToMany(Role::class)
    ->wherePivotNull('expired_at'); // إضافة شرط حيث يكون العمود فارغًا

return $this->belongsToMany(Role::class)
```

```
->wherePivotNotNull('expired_at'); // إضافة شرط حيث يكون العمود غير فارغ
```

ترتيب الاستعلامات عبر أعمدة الجدول الوسيط(Ordering Queries via Intermediate Table Columns)

يمكنك ترتيب النتائج التي تم إرجاعها بواسطة استعلامات العلاقة belongToMany باستخدام الدالة orderByPivot.

في المثال التالي، سنقوم باسترداد جميع الشارات الأحدث للمستخدم:

```
return $this->belongsToMany(Badge::class)
        ->where('rank', 'gold')
        ->orderByPivot('created_at', 'desc'); // ترتيب حسب العمود الوسيط
```

تعريف نماذج الجداول الوسيطة المخصصة(Defining Custom Intermediate Table Models)

إذا كنت ترغب في تعريف نموذج مخصص لتمثيل الجدول الوسيط لعلاقتك المتعددة إلى المتعددة، فيمكنك استدعاء الدالة using عند تعريف العلاقة.

تتيح لك الـpivot models المخصصة الفرصة لتعريف سلوك إضافي على الـpivot model ، مثل الدوال والتحويلات.

يجب أن ترث الـpivot models المخصصة والتي تمثل علاقة متعدد إلى متعدد من الفئة Illuminate\Database\Eloquent\Relations\Pivot بينما يجب أن ترث الـpolymorphic pivot models المخصصة والتي تمثل علاقة متعدد إلى متعدد من الفئة Illuminate\Database\Eloquent\Relations\MorphPivot.

على سبيل المثال، يمكننا تعريف نموذج دور يستخدم نموذج محور مخصص من RoleUser:

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

class Role extends Model
{
   /**
    * The users that belong to the role.     * المستخدمون التابعون للدور
    */
   public function users(): BelongsToMany
   {
      return $this->belongsToMany(User::class)->using(RoleUser::class); //  لجدول الربط (RoleUser)تحديد موديل مخصص
   }
}
```

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot
{
   // ...
}
```

لا يجوز لـPivot models استخدام خاصية SoftDeletes.

إذا كنت بحاجة إلى حذف سجلات Pivot بشكل مؤقت، ففكر في تحويل نموذج Pivot الخاص بك إلى نموذج Eloquent فعلي.

النماذج الوسيطة المخصصة والمفاتيح المتزايدة(Custom Pivot Models and Incrementing IDs)

إذا قمت بتعريف علاقة متعدد الى متعدد تستخدم pivot model مخصص، وكان هذا الـpivot model يحتوي على مفتاح أساسي متزايد تلقائيًا، فيجب عليك التأكد من أن فئة الـ pivot model المخصص لديك تحدد او تسند القيمة pivot model للخاصية $incrementing.

```
/**
 * Indicates if the IDs are auto-incrementing.     * تشير إلى ما إذا كانت المعرفات تزداد تلقائياً
 *
 * @var bool
 */
   public $incrementing = true; // تفعيل المفاتيح المتزايدة تلقائياً
```

العلاقات متعددة الأشكال (Polymorphic Relationships)

تسمح علاقة الـpolymorphic للـmodel الفرعي بالانتماء إلى أكثر من نوع من النماذج باستخدام ارتباط واحد. على سبيل المثال، تخيل أنك تقوم ببناء تطبيق يسمح للمستخدمين بمشاركة منشورات المدونة ومقاطع الفيديو.

في مثل هذا التطبيق، قد ينتمي modelالتعليق إلى كل من modelالمنشور و modelالفيديو.

العلاقة واحد لواحد متعددة الأشكال - One to One (Polymorphic)

هيكل الجدول (Table Structure)

العلاقة polymorphic التي تمثل واحد إلى واحد تشبه العلاقة النموذجية من واحد إلى واحد؛ ومع ذلك، يمكن أن ينتمي النموذج model إلى أكثر من نوع من model باستخدام ارتباط واحد.

على سبيل المثال، قد تشترك منشور مدونة ومستخدم في علاقة الـpolymorphic مع model صورة.

يتيح لك استخدام علاقة polymorphic من واحد إلى واحد الحصول على جدول واحد من الصور الفريدة التي يمكن ربطها بالمشاركات والمستخدمين. أولاً، دعنا نفحص بنية الجدول:

```
posts
   id - integer
   name - string

users
   id - integer
   name - string

images
   id - integer
   url - string
   imageable_id - integer // معرف النموذج الأب
   imageable_type - string // نوع النموذج الأب
```

لاحظ الأعمدة imageable_id و imageable_type في جدول الصور.

سيحتوي عمود imageable_id على قيمة معرف المنشور أو المستخدم، بينما سيحتوي عمود imageable_type على اسم فئة النموذج الأساسي.

يستخدم Eloquent عمود imageable_type لتحديد "نوع" النموذج الأساسي الذي سيتم إرجاعه عند الوصول إلى علاقة imageable.

في هذه الحالة، سيحتوي العمود إما على App\Models\Post أو App\Models\User.

هيكل النماذج (Model Structure)

والآن، دعونا نتفحص تعريفات النماذج اللازمة لبناء هذه العلاقة:

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

class Image extends Model
{
   /**
    * Get the parent imageable model (user or post).     * الحصول على النموذج الأب القابل للربط بالصورة (مستخدم أو منشور)
    */
   public function imageable(): MorphTo
   {
      return $this->morphOne(Image::class, 'imageable'); // علاقة واحدة متعددة
   }
}
```

```php
 use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;

class Post extends Model
{
   /**
    * Get the post's image.     * الحصول على صورة المنشور
    */
   public function image(): MorphOne
   {
      return $this->morphOne(Image::class, 'imageable'); // علاقة واحدة متعددة
   }
}
```

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

class User extends Model
{
   /**
    * Get the user's image.     * الحصول على صورة المستخدم
    */
   public function image(): MorphOne
   {
      return $this->morphOne(Image::class, 'imageable');
   }
}
```

استرجاع العلاقة(Retrieving the Relationship)

بمجرد تحديد جدول قاعدة البيانات والنماذج، يمكنك الوصول إلى العلاقات عبر النماذج الخاصة بك. على سبيل المثال، لاسترداد الصورة لمنشور، يمكننا الوصول إلى خاصية العلاقة الديناميكية للصورة:

```
use App\Models\Post;

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

يمكنك استرداد أصل النموذج المتعدد الأشكال من خلال الوصول إلى اسم الدالة التي تقوم باستدعاء.morphTo

في هذه الحالة، هذه هي الطريقة التي يمكن إنشاء صورة لها في نموذج.Image

لذا، سنصل إلى هذه الدالة كخاصية علاقة ديناميكية:

```
use App\Models\Image;
$image = Image::find(1);
$imageable = $image->imageable;
```

ستعيد العلاقة imageable في نموذج الصورة إما مثيل Post أو User، اعتمادًا على نوع النموذج الذي يمتلك الصورة.

الاتفاقيات الأساسية(Key Conventions)

إذا لزم الأمر، يمكنك تحديد اسم عمودي "id" و"type" اللذين يستخدمهما نموذج الفرعي المتعدد الأشكال.

إذا قمت بذلك، فتأكد من تمرير اسم العلاقة دائمًا كمعامل أول إلى الدالة morphTo. عادةً، يجب أن تتطابق هذه القيمة مع اسم الدالة، لذا يمكنك استخدام ثابت PHP __FUNCTION__:

```
/**
 * Get the model that the image belongs to.     * الحصول على النموذج الذي تنتمي إليه الصورة
 */
public function imageable(): MorphTo
{
   return $this->morphTo(__FUNCTION__, 'imageable_type', 'imageable_id');
}
```

One to Many (Polymorphic) العلاقة واحد لكثير متعددة الأشكال -

هيكل الجداول(Table Structure)

تتشابه العلاقة المتعددة الأشكال من واحد إلى كثير مع العلاقة النموذجية من واحد إلى كثير؛ ومع ذلك، يمكن أن ينتمي النموذج الفرعي إلى أكثر من نوع من النماذج باستخدام ارتباط واحد.

على سبيل المثال، تخيل أن مستخدمي تطبيقك يمكنهم "التعليق" على المنشورات ومقاطع الفيديو.

باستخدام العلاقات متعددة الأشكال، يمكنك استخدام جدول تعليقات واحد لاحتواء التعليقات على كل من المنشورات ومقاطع الفيديو.

أولاً، دعنا نفحص بنية الجدول المطلوبة لبناء هذه العلاقة:

```
posts
   id - integer
   title - string
   body - text

videos
```

```
    id - integer
    title - string
    url - string

comments
    id - integer
    body - text
    commentable_id - integer   -- معرف النموذج القابل للتعليق
    commentable_type - string  -- نوع النموذج القابل للتعليق
```

هيكل النماذج(Model Structure)

والآن، دعونا نتفحص تعريفات النموذج اللازمة لبناء هذه العلاقة:

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

class Comment extends Model
{
    /**
     * Get the parent commentable model (post or video).     * الحصول على النموذج الأب القابل للتعليق (منشور أو فيديو)
     */
    public function commentable(): MorphTo
    {
        return $this->morphTo();// علاقة متعددة الأشكال
    }
}
```

```php
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Post extends Model
{
    /**
     * Get all of the post's comments.     * الحصول على جميع تعليقات المنشور
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable'); // علاقة كثيرة متعددة
    }
}
```

```php
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Video extends Model
{
    /**
     * Get all of the video's comments.     * الحصول على جميع تعليقات الفيديو
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable'); // علاقة كثيرة متعددة
```

```
    }
}
```

استرجاع العلاقة(Retrieving the Relationship)

بمجرد تعريف جدول قاعدة البيانات والنماذج، يمكنك الوصول إلى العلاقات عبر خصائص العلاقة الديناميكية للنموذج. على سبيل المثال، للوصول إلى جميع التعليقات على منشور، يمكننا استخدام خاصية التعليقات الديناميكية:

```
use App\Models\Post;
$post = Post::find(1);
foreach ($post->comments as $comment) {
    // ...
}
```

يمكنك أيضًا استرداد الـ parent model للـ child model في علاقة الـpolymorphic من خلال الوصول إلى اسم الدالة التي تقوم بإجراء الاستدعاء للدالة morphTo.

في هذه الحالة، هذه الدالة هي commentable المعرفة في الفئة Comment.

لذا، سنصل إلى هذه الدالة كخاصية علاقة ديناميكية من أجل الوصول إلى الـ parent model للتعليق:

```
use App\Models\Comment;
$comment = Comment::find(1);
$commentable = $comment->commentable;
```

ستعيد العلاقة commentable في نموذج Comment إما كائن منشور أو فيديو، اعتمادًا على نوع الـmodel الذي يمثل parent model للتعليق.

التحميل التلقائي للنماذج الأب على النماذج الفرعية(Automatically Hydrating Parent Models on Children)

حتى عند استخدام التحميل السريع لـ Eloquent، قد تنشأ مشكلات استعلام "N + 1" إذا حاولت الوصول إلى parent model من child model أثناء التكرار عبر الـchild models :

```
$posts = Post::with('comments')->get(); // التحميل المسبق للتعليقات
foreach ($posts as $post) {
    foreach ($post->comments as $comment) {
        echo $comment->commentable->title;
    }
}
```

في المثال أعلاه، تم تقديم مشكلة استعلام "N + 1" لأنه، على الرغم من تحميل التعليقات بشغف لكل نموذج Post، فإن Eloquent لا يقوم تلقائيًا بترطيب Post الرئيسي في كل نموذج تعليق فرعي.

إذا كنت ترغب في أن يقوم Eloquent تلقائيًا بترطيب النماذج الأصلية على أطفالها، فيمكنك استدعاء طريقة chaperone عند تعريف علاقة morphMany:

```
class Post extends Model
{
    /**
     * Get all of the post's comments.     * الحصول على جميع تعليقات المنشور
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable')->chaperone();// علاقة كثيرة مع تحميل تلقائي للنموذج الأب
    }
}
```

أو، إذا كنت ترغب في الاشتراك في الترطيب التلقائي للوالدين في وقت التشغيل، فيمكنك استدعاء نموذج المرافق عند تحميل العلاقة بشغف:

```
use App\Models\Post;
$posts = Post::with([
    'comments' => fn ($comments) => $comments->chaperone(),    // تحميل تلقائي ديناميكي للنماذج الأب
])->get();
```

التفويض:(Authorization)

بالإضافة إلى توفير خدمات المصادقة المضمنة، يوفر Laravel أيضًا طريقة بسيطة لتفويض إجراءات المستخدم على مورد معين. على سبيل المثال، حتى لو تم مصادقة المستخدم، فقد لا يُسمح له بتحديث أو حذف نماذج Eloquent معينة أو سجلات قاعدة البيانات التي يديرها التطبيق. توفر ميزات التفويض في Laravel طريقة سهلة ومنظمة لإدارة هذه الأنواع من عمليات التحقق من التفويض.

يوفر Laravel طريقتين أساسيتين لتفويض الإجراءات: الـgates والـ policies. فكر في الـgates والـ policies. مثل الـ routes والـ controllers.

توفر الـGates نهجًا بسيطًا قائمًا على استخدام الـ closure-based للتفويض بينما تقوم الـpolicies ، مثل الـcontrollers ، بتجميع المنطق حول modelأو resource معين.

لا تحتاج إلى الاختيار بين استخدام البوابات حصريًا أو استخدام السياسات حصريًا عند إنشاء تطبيق. من المرجح أن تحتوي معظم التطبيقات على مزيج من البوابات والسياسات، وهذا أمر جيد تمامًا! البوابات قابلة للتطبيق بشكل أكبر على الإجراءات غير المرتبطة بأي نموذج أو مورد، مثل عرض لوحة معلومات المسؤول. على النقيض من ذلك، يجب استخدام السياسات عندما ترغب في تفويض إجراء لنموذج أو مورد معين.

البوابات(Gates)

كتابة البوابات(Writing Gates)

البوابات هي ببساطة إغلاقات تحدد ما إذا كان المستخدم مخولاً بأداء إجراء معين. عادةً، يتم تعريف البوابات داخل دالة boot لفئة App\Providers\AppServiceProvider باستخدام واجهة Gate. تتلقى البوابات دائمًا مثيل مستخدم كمعامل أولى ويمكنها بشكل اختياري تلقي معاملات إضافية مثل نموذج Eloquent ذي الصلة.

في هذا المثال، سنقوم بتعريف بوابة لتحديد ما إذا كان بإمكان المستخدم تحديث نموذج App\Models\Post معين. ستنجز البوابة ذلك من خلال مقارنة معرف المستخدم بمعرف المستخدم الذي أنشأ المنشور:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * Bootstrap any application services.     * تسجيل أي خدمات للتطبيق
 */
public function boot(): void
{
   Gate::define('update-post', function (User $user, Post $post) {
      return $user->id === $post->user_id;
   });
}
```

مثل وحدات التحكم، يمكن أيضًا تعريف البوابات باستخدام مصفوفة استدعاء للفئة:

```
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;

/**
 * Bootstrap any application services.     * تسجيل أي خدمات للتطبيق
 */
public function boot(): void
{
   Gate::define('update-post', [PostPolicy::class, 'update']);
}
```

تفويض الإجراءات(Authorizing Actions)

لتفويض إجراء باستخدام البوابات، يجب عليك استخدام دوال السماح أو الرفض التي توفرها واجهة البوابة. لاحظ أنه لا يلزمك تمرير المستخدم المعتمد حاليًا إلى هذه الدوال. سيتولى Laravel تلقائيًا تمرير المستخدم إلى إغلاق البوابة. من المعتاد استدعاء دوال تفويض البوابة داخل وحدات التحكم في التطبيق قبل تنفيذ إجراء يتطلب تفويضًا:

```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class PostController extends Controller
```

```
{
    /**
     * Update the given post.     * تحديث المنشور المحدد
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        if (! Gate::allows('update-post', $post)) {        // Gate التحقق من الصلاحية باستخدام
            abort(403);          // خطأ غير مصرح به
        }
        // Update the post...        // تحديث المنشور...
        return redirect('/posts');
    }
}
```

إذا كنت ترغب في تحديد ما إذا كان مستخدم آخر غير المستخدم المعتمد حاليًا مخولًا لأداء إجراء، فيمكنك استخدام دالة forUser على واجهة Gate:

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    // The user can update the post...    // المستخدم يمكنه تحديث المنشور...
}'update
if (Gate::forUser($user)->denies(-post', $post)) {
    // The user can't update the post...    // المستخدم لا يمكنه تحديث المنشور...
}
```

يمكنك تفويض إجراءات متعددة في وقت واحد باستخدام الدالة any او الدالة none:

```
if (Gate::any(['update-post', 'delete-post'], $post)) {
    // The user can update or delete the post...    // المستخدم يمكنه تحديث أو حذف المنشور...
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // The user can't update or delete the post...    // المستخدم لا يمكنه تحديث ولا حذف المنشور...
}
```

التفويض أو إلقاء الاستثناءات(Authorizing or Throwing Exceptions)

إذا كنت ترغب في محاولة تفويض إجراء وإلقاء Illuminate\Auth\Access\AuthorizationException تلقائيًا إذا لم يُسمح للمستخدم بتنفيذ الإجراء المحدد، فيمكنك استخدام الدالة Authorize في واجهة Gate. يتم تحويل حالات AuthorizationException تلقائيًا إلى استجابة HTTP 403 بواسطة Laravel:

```
Gate::authorize('update-post', $post);
// The action is authorized... // ... يستمر الكود إذا كان مصرحًا
```

توفير سياق إضافي(Supplying Additional Context)

دوال بوابة التفويض هي (allows, denies, check, any, none, authorize, can, cannot) و Blade directives التفويض هي (@can، @cannot، @canany) تستقبل مصفوفة كمعامل ثاني لها. يتم تمرير هذه المصفوفة كمعاملات إلى دالة إغلاق البوابة، ويمكن استخدامها كسياق مضاف عند اتخاذ قرارات التفويض:

```
use App\Models\Category;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::define('create-post', function (User $user, Category $category, bool $pinned) {
    if (! $user->canPublishToGroup($category->group)) {    // منطق التحقق باستخدام المعاملات الإضافية
        return false;
    } elseif ($pinned && ! $user->canPinPosts()) {
        return false; // استدعاء البوابة مع تمرير معاملات إضافية في مصفوفة
    }
    return true;
});

if (Gate::check('create-post', [$category, $pinned])) {
    // The user can create the post...    // المستخدم يمكنه إنشاء المنشور...
}
```

استجابات البوابة(Gate Responses)

حتى الآن، قمنا فقط بفحص البوابات التي تعيد قيمًا منطقية بسيطة. ومع ذلك، قد ترغب أحيانًا في إرجاع استجابة أكثر تفصيلاً، بما في ذلك رسالة خطأ. للقيام بذلك، يمكنك إرجاع Illuminate\Auth\Access\Response من البوابة الخاصة بك:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;
Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
        ? Response::allow() // الإذن مسموح
        : Response::deny('You must be an administrator.'); // الإذن مرفوق برسالة
});
```

حتى عندما تقوم بإرجاع استجابة الترخيص من البوابة الخاصة بك، فإن الدالة Gate::allows ستظل تعيد قيمة منطقية بسيطة؛ ومع ذلك، يمكنك استخدام الدالة Gate::inspect للحصول على استجابة التفويض الكاملة التي تم إرجاعها بواسطة البوابة:

```
$response = Gate::inspect('edit-settings');
if ($response->allowed()) {
    // The action is authorized... // الإجراء مصرح به...
} else {
    echo $response->message();// عرض رسالة الخطأ من الاستجابة
}
```

عند استخدام الدالة Gate::authorize، التي ترمي AuthorizationException إذا لم يتم تفويض الإجراء، سيتم نشر رسالة الخطأ المقدمة بواسطة استجابة التفويض إلى استجابة HTTP:

```
Gate::authorize('edit-settings');
// The action is authorized...        // الإجراء مصرح به...
```

تخصيص حالة استجابة(Customizing The HTTP Response Status) HTTP

عند رفض إجراء عبر بوابة، يتم إرجاع استجابة HTTP 403؛ ومع ذلك، قد يكون من المفيد في بعض الأحيان إرجاع رمز حالة HTTP بديل. يمكنك تخصيص رمز حالة HTTP الذي يتم إرجاعه لفحص تفويض فاشل باستخدام البناء denyWithStatus الساكنة في الفئة Illuminate\Auth\Access\Response:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
            ? Response::allow()
            : Response::denyWithStatus(404); // إرجاع خطأ ٤٠٤ بدلاً من ٤٠٣ عند الرفض
});
```

نظرًا لأن إخفاء الموارد عبر استجابة ٤٠٤ يعد نمطًا شائعًا لتطبيقات الويب، يتم استخدام الدالة denyAsNotFound للتسهيل:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;
Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
            ? Response::allow()
            : Response::denyAsNotFound();// عند الرفض (Not Found) إرجاع خطأ ٤٠٤
});
```

اعتراض فحوصات البوابة(Intercepting Gate Checks)

في بعض الأحيان، قد ترغب في منح جميع القدرات (الصلاحيات) لمستخدم معين. يمكنك استخدام الدالة before لتعريف إغلاق يتم تشغيله قبل جميع عمليات فحص التفويض الأخرى:

```
use App\Models\User;
use Illuminate\Support\Facades\Gate;
Gate::before(function (User $user, string $ability) {// قبل أي فحص تفويض آخر Closure يتم تنفيذ هذا الـ
    if ($user->isAdministrator()) {    // إذا كان المستخدم مديرًا، اسمح له بكل شيء
        return true;
    }
});
```

إذا أعاد الـ closure السابق نتيجة non-null، فسيتم اعتبار هذه النتيجة نتيجة فحص التفويض.

يمكنك استخدام الدالة after لتعريف closure سيتم تنفيذه بعد جميع عمليات فحص التفويض الأخرى:

```
use App\Models\User;

Gate::after(function (User $user, string $ability, bool|null $result, mixed $arguments) {
   if ($user->isAdministrator()) {
      return true;
   }
});
```

القيم التي يتم إرجاعها بعد تنفيذ عمليات closures التي تم تعريفها للدالة after لن تحل محل نتيجة فحص الترخيص إلا إذا أرجعت gate أو policy قيمة null.

التفويض المضمن(Inline Authorization)

التفويض المضمن
في بعض الأحيان، قد ترغب في تحديد ما إذا كان المستخدم المعتمد حاليًا مخولًا لأداء إجراء معين دون كتابة gate مخصصة تتوافق مع الإجراء.
يتيح لك Laravel إجراء هذه الأنواع من عمليات التحقق من التفويض "المضمن" عبر الدالتين Gate::allowIf وGate::denyIf. لا ينفذ التفويض المضمن أي before authorization hooks أو after authorization hooks محددة:

```
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::allowIf(fn (User $user) => $user->isAdministrator());// السماح للمسؤولين فقط (تفويض مضمن)
Gate::denyIf(fn (User $user) => $user->banned());// رفض المستخدمين المحظورين (تفويض مضمن)
```

إذا لم يتم تفويض الإجراء أو إذا لم يتم مصادقة أي مستخدم حاليًا، فسوف يرمي Laravel تلقائيًا استثناء Illuminate\Auth\Access\AuthorizationException.
يتم تحويل حالات AuthorizationException تلقائيًا إلى استجابة HTTP 403 بواسطة معالج استثناء Laravel.

إنشاء السياسات(Creating Policies)

إنشاء السياسات(Generating Policies)

السياسات عبارة عن فئات تنظم منطق التفويض حول نموذج أو مورد معين. على سبيل المثال، إذا كان تطبيقك عبارة عن مدونة، فقد يكون لديك نموذج App\Models\Post ونموذج App\Policies\PostPolicy المقابل لتفويض إجراءات المستخدم مثل إنشاء أو تحديث المنشورات.
يمكنك إنشاء policy باستخدام الأمر artisan make:policy. سيتم وضع السياسة المولدة في دليل app/Policies. إذا لم يكن هذا الدليل موجودًا في تطبيقك، فسيقوم Laravel بإنشائه لك:

```
php artisan make:policy PostPolicy
```

سيعمل الأمر make:policy على إنشاء فئة policy فارغة. إذا كنت ترغب في إنشاء فئة تحتوي على داول policy نموذجية تتعلق بعرض الموارد وإنشائها وتحديثها وحذفها، فيمكنك توفير خيار model-- عند تنفيذ الأمر:

```
php artisan make:policy PostPolicy --model=Post
```

تسجيل السياسات(Registering Policies)

اكتشاف السياسات(Policy Discovery)

بشكل افتراضي، يقوم Laravel باكتشاف سياسات الصلاحية (policies) تلقائيًا طالما أن النموذج (model) والسياسة (policy) يتبعان اصطلاحات التسمية القياسية فيLaravel . على وجه التحديد، يجب أن تكون السياسات موجودة في مجلد اسمه Policies، وهذا المجلد يجب أن يكون في نفس مستوى مجلد النماذج (Models)أو في مستوى أعلى منه.
لذلك، على سبيل المثال، يمكن وضع النماذج (models) في مجلد app/Models بينما يمكن وضع السياسات (policies) في مجلد app/Policies. في هذه الحالة، سيبحث Laravel عن السياسات في app/Models/Policies ثم في app/Policies.
بالإضافة إلى ذلك، يجب أن يتطابق اسم السياسة مع اسم النموذج وأن يحتوي على لاحقة سياسة. لذا، فإن نموذج المستخدم يتوافق مع فئة سياسة UserPolicy.
إذا كنت ترغب في تعريف منطق خاص بك لاكتشاف سياسات الصلاحية(policy discovery logic) ، يمكنك تسجيل دالة (callback)مخصصة لاكتشاف السياسات باستخدام دالة Gate::guessPolicyNamesUsing. عادةً، يجب استدعاء هذه الدالة من داخل دالة bootالموجودة في ملف AppServiceProviderالخاص بتطبيقك.

```
use Illuminate\Support\Facades\Gate;
Gate::guessPolicyNamesUsing(function (string $modelClass) {
   // Return the name of the policy class for the given model... // ... منطق مخصص لإرجاع اسم Policy بناءً على اسم الـ Model
});
```

التسجيل اليدوي للسياسات(Manually Registering Policies)

باستخدام (Gate facade)، يمكنك تسجيل سياسات الصلاحية (policies) والنماذج (models) المرتبطة بها يدويًا داخل دالة bootالموجودة في ملف AppServiceProviderالخاص بتطبيقك.

```
use App\Models\Order;
```

```
use App\Policies\OrderPolicy;
use Illuminate\Support\Facades\Gate;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
   Gate::policy(Order::class, OrderPolicy::class);    // تسجيل Policy يدويًا لنموذج Order
}
```

كتابة السياسات(Writing Policies)

دوال السياسة(Policy Methods)

بمجرد تسجيل فئة السياسة(policy class) ، يمكنك إضافة دوال (methods) لكل إجراء (action) تمنح صلاحيته. على سبيل المثال، دعنا نُعرّف دالة update في سياسة PostPolicy الخاصة بنا، والتي تحدد ما إذا كان مستخدم معين (App\Models\User) يمكنه تحديث تدوينة معينة(App\Models\Post) .
ستستقبل دالة update كائنًا من User وكائنًا من Post كوسائط لها، ويجب أن تُعيد true أو false للإشارة إلى ما إذا كان المستخدم مصرحًا له بتحديث كائن Post المحدد. لذلك، في هذا المثال، سنتحقق من أن مُعرّف المستخدم (id) يطابق حقل user_id الموجود في التدوينة.

```
<?php
namespace App\Policies;
use App\Models\Post;
use App\Models\User;

class PostPolicy
{
   /**
    * Determine if the given post can be updated by the user.     * تحديد ما إذا كان يمكن للمستخدم تحديث المنشور المحدد
    */
   public function update(User $user, Post $post): bool
   {
      return $user->id === $post->user_id;       // التحقق من أن المستخدم هو مالك المنشور

   }
}
```

يمكنك الاستمرار في تعريف دوال إضافية في السياسة (policy) حسب الحاجة لمختلف الإجراءات (actions) التي تمنح صلاحيتها. على سبيل المثال، قد تُعرّف دالتي view(عرض) أو delete(حذف) لمنح صلاحيات لإجراءات متنوعة متعلقة بالتدوينات(Post) ، لكن تذكر أن لديك الحرية في إعطاء دوال السياسة الخاصة بك أي اسم تريده.
إذا استخدمت خيار model-- عند إنشاء سياستك عبر شاشة اوامر Artisan، فسوف يحتوي بالفعل على دوال لإجراءات viewAny وview وcreate وupdate وdelete وrestore وforceDelete.

استجابات السياسة(Policy Responses)

حتى الآن، لقد قمنا فقط بفحص دوال السياسة (policy methods) التي تُعيد قيمًا منطقية بسيطة(boolean values) . ومع ذلك، في بعض الأحيان قد ترغب في إعادة استجابة أكثر تفصيلاً، تتضمن رسالة خطأ. للقيام بذلك، يمكنك إعادة كائن من Illuminate\Auth\Access\Response من دالة السياسة الخاصة بك.

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;
/**
 * Determine if the given post can be updated by the user.     * تحديد ما إذا كان يمكن للمستخدم تحديث المنشور المحدد
 */
public function update(User $user, Post $post): Response
{
   return $user->id === $post->user_id
            ? Response::allow()
            : Response::deny('You do not own this post.'); // ممنوع مع رسالة
}
```

عند إعادة استجابة صلاحية (authorization response) من سياستك، ستظل دالة Gate::allows تُعيد قيمة منطقية بسيطة(boolean value) ؛ ومع ذلك، يمكنك استخدام دالة Gate::inspect للحصول على استجابة الصلاحية الكاملة التي أعادتها البوابة(gate) .

```
use Illuminate\Support\Facades\Gate;
$response = Gate::inspect('update', $post); // فحص الاستجابة الكاملة من Policy
if ($response->allowed()) {
   // The action is authorized... // الإجراء مصرح به...
} else {
   echo $response->message(); // عرض رسالة الخطأ
}
```

عند استخدام دالة Gate::authorize، والتي تقوم بإلقاء استثناء من نوع AuthorizationExceptionإذا كان الإجراء غير مصرح به، فإن رسالة الخطأ التي تم توفيرها في استجابة الصلاحية (authorization response) سيتم نشرها (propagated) إلى استجابة الـHTTP .

```
Gate::authorize('update', $post);
// The action is authorized...     // الإجراء مصرح به...
```

تخصيص حالة استجابةHTTP (Customizing the HTTP Response Status)

عندما يتم رفض إجراء (action) عبر دالة سياسة(policy method) ، يتم إعادة استجابة HTTP برمز الحالة ٤٠٣ ومع ذلك، قد يكون من المفيد أحيانًا إعادة رمز حالة HTTP بديل. يمكنك تخصيص رمز حالة HTTP المُعاد عند فشل التحقق من الصلاحية باستخدام دالة البناء الساكنة denyWithStatus الموجود في الفئة Illuminate\Auth\Access\Response.

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;
/**
 * Determine if the given post can be updated by the user.     * تحديد ما إذا كان يمكن للمستخدم تحديث المنشور المحدد
 */
public function update(User $user, Post $post): Response
{
   return $user->id === $post->user_id
            ? Response::allow()
            : Response::denyWithStatus(404);
}
```

نظرًا لأن إخفاء الموارد عبر استجابة ٤٠٤ يعد نمطًا شائعًا لتطبيقات الويب، يتم تقديم دالة denyAsNotFound للتسهيل على المطورين:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Determine if the given post can be updated by the user.     * تحديد ما إذا كان يمكن للمستخدم تحديث المنشور المحدد
 */
public function update(User $user, Post $post): Response
{
   return $user->id === $post->user_id
            ? Response::allow()
            : Response::denyAsNotFound();
}
```

الدوال التي لا تتطلب نماذج(Methods Without Models)

بعض دوال السياسة (policy methods) تستقبل فقط كائنًا للمستخدم المُصادق عليه حاليًا(currently authenticated user) . هذا الموقف هو الأكثر شيوعًا عند منح صلاحية لإجراءات الـ (create) . على سبيل المثال، إذا كنت تقوم بإنشاء مدونة، فقد ترغب في تحديد ما إذا كان المستخدم مصرحًا له بإنشاء أي تدوينات على الإطلاق. في هذه الحالات، يجب أن تتوقع دالة السياسة الخاصة بك أن تستقبل كائن المستخدم فقط.

```
/**
 * Determine if the given user can create posts. * تحديد ما إذا كان يمكن للمستخدم إنشاء المنشورات
 */
public function create(User $user): bool
{
   return $user->role == 'writer';    // التحقق من دور المستخدم قبل السماح له بإنشاء منشورات
}
```

المستخدمون الضيوف(Guest Users)

بشكل افتراضي، جميع البوابات (gates) والسياسات (policies) تُعيد falseتلقائيًا إذا لم يكن طلب HTTP الوارد قد بدأ بواسطة مستخدم مُصادق عليه (authenticated user). ومع ذلك، يمكنك السماح لهذه الفحوصات بالمرور إلى البوابات والسياسات الخاصة بك عن طريق تعريف تلميح النوع (type-hint) كـ "اختياري (optional) "أو عن طريق توفير قيمة افتراضية nullلتعريف وسيط المستخدم.

```
<?php
 namespace App\Policies;
 use App\Models\Post;
use App\Models\User;

class PostPolicy
{
   /**
    * Determine if the given post can be updated by the user.     * تحديد ما إذا كان يمكن للمستخدم تحديث المنشور المحدد
    */
   public function update(?User $user, Post $post): bool
   {
      return $user?->id === $post->user_id;        // استخدام Null Safe Operator للتحقق إذا كان المستخدم موجودًا
   }
}
```

مرشحات السياسة(Policy Filters)

في السياسة. سيتم تنفيذ beforeبالنسبة لمستخدمين معينين، قد ترغب في منحهم صلاحية لجميع الإجراءات داخل سياسة معينة. لتحقيق ذلك، قم بتعريف دالة قبل أي دوال أخرى في السياسة، مما يمنحك فرصة لمنح الصلاحية للإجراء قبل أن يتم استدعاء دالة السياسة المقصودة فعليًا. هذه الميزة هي beforeدالة صلاحية لأداء أي إجراء (administrators) الأكثر استخدامًا لمنح مديري التطبيق.

```
use App\Models\User;

/**
 * Perform pre-authorization checks. * تنفيذ فحوصات التفويض المسبقة
 */
public function before(User $user, string $ability): bool|null
{
   if ($user->isAdministrator()) {    // إذا كان المستخدم مديرًا، اسمح له بكل الإجراءات
      return true;
   }
   return null; // إذا لم يكن مديرًا، اترك القرار للدالة الخاصة بالإجراء
}
```

إذا كنت ترغب في رفض جميع عمليات التحقق من التفويض لنوع معين من المستخدمين، فيمكنك إرجاع القيمة false من الدالة before. إذا تم إرجاع القيمة null، فسوف تنتقل عملية التحقق من التفويض إلى دالة السياسة.

تفويض الإجراءات باستخدام السياسات(Authorizing Actions Using Policies)

عبر نموذج المستخدم(Via the User Model)

نموذج App\Models\Userالذي يأتي مع تطبيق Laravel الخاص بك يتضمن دالتين مساعدتين (helpful methods) لمنح الصلاحيات للإجراءات can : و cannot. تستقبل دالتا canو cannotاسم الإجراء الذي ترغب في التحقق من صلاحيته والنموذج ذي الصلة. على سبيل المثال، دعنا نحدد ما إذا كان مستخدم معين مصرحًا له بتحديث نموذج App\Models\Postمعين. عادةً، يتم ذلك داخل دالة في المتحكم(controller method) .

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
class PostController extends Controller
{
   /**
    * Update the given post. * تحديث المنشور المحدد
    */
```

```
    public function update(Request $request, Post $post): RedirectResponse
    {        // استخدام دالة cannot للتحقق من الصلاحية المساعدة على نموذج المستخدم
        if ($request->user()->cannot('update', $post)) {
            abort(403);
        }
        // Update the post...        // ... كود التحديث
        return redirect('/posts');
    }
}
```

إذا تم تسجيل سياسة للنموذج المحدد، فستقوم دالة can تلقائيًا باستدعاء السياسة المناسبة وإرجاع النتيجة المنطقية. إذا لم يتم تسجيل أي سياسة للنموذج، فستحاول الدالة can استدعاء البوابة القائمة على الإغلاق المطابقة لاسم الإجراء المحدد.

الإجراءات التي لا تتطلب نماذج(Actions That Don't Require Models)

تذكر أن بعض الإجراءات قد تتوافق مع أساليب السياسة مثل الإنشاء التي لا تتطلب مثيل نموذج. في هذه المواقف، يمكنك تمرير اسم فئة إلى دالة can. سيتم استخدام اسم الفئة لتحديد السياسة التي سيتم استخدامها عند تفويض الإجراء:

```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Create a post.
     */
    public function store(Request $request): RedirectResponse
    {        // التحقق من صلاحية إنشاء منشور (إجراء لا يتطلب مثيل نموذج)
        if ($request->user()->cannot('create', Post::class)) {
            abort(403);
        }
        // Create the post...        // ... كود الإنشاء
        return redirect('/posts');
    }
}
```

عبر واجهة البوابة(Via the Gate Facade)

عبر واجهة البوابة

بالإضافة إلى الدوال المفيدة المقدمة لنموذج App\Models\User، يمكنك دائمًا تفويض الإجراءات عبر دالة تفويض واجهة البوابة.

مثل الدالة can، تقبل هذه الدالة اسم الإجراء الذي ترغب في تفويضه والنموذج ذي الصلة. إذا لم يتم تفويض الإجراء، فستلقي دالة authorize استثناء Illuminate\Auth\Access\AuthorizationException والذي سيحوله معالج استثناء Laravel تلقائيًا إلى استجابة HTTP برمز حالة ٤٠٣:

```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class PostController extends Controller
{
    /**
     * Update the given blog post.
     *
     * @throws \Illuminate\Auth\Access\AuthorizationException
```

```
    */
    public function update(Request $request, Post $post): RedirectResponse
    {
        Gate::authorize('update', $post);              // تفويض الإجراء باستخدام Gate facade - يرمي استثناء تلقائيًا إذا فشل
        // The current user can update the blog post...          // ... كود التحديث

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

الإجراءات التي لا تتطلب نماذج(Actions That Don't Require Models)

كما ناقشنا سابقًا، لا تتطلب بعض دوال السياسة مثل الـ create مثيلًا للنموذج. في هذه المواقف، يجب عليك تمرير اسم فئة إلى الدالة authorize. سيتم استخدام اسم الفئة لتحديد السياسة التي سيتم استخدامها عند تفويض الإجراء:

```
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

/**
 * Create a new blog post.
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function create(Request $request): RedirectResponse
{
    Gate::authorize('create', Post::class);    // تفويض إجراء الإنشاء (لا يتطلب مثيل نموذج)
    // The current user can create blog posts...    // ... كود الإنشاء
    return redirect('/posts');
}
```

عبر البرنامج الوسيط(Middleware)

يتضمن Laravel برنامجًا وسيطًا يمكنه تفويض الإجراءات قبل أن يصل الطلب الوارد إلى مساراتك أو وحدات التحكم الخاصة بك. بشكل افتراضي، يمكن ربط البرنامج الوسيط Illuminate\Auth\Middleware\Authorize بمسار باستخدام الاسم المستعار للبرنامج الوسيط can، والذي يتم تسجيله تلقائيًا بواسطة Laravel. دعنا نستكشف مثالاً لاستخدام البرنامج الوسيط can لتفويض مستخدم بتحديث منشور:

```
use App\Models\Post;
Route::put('/post/{post}', function (Post $post) {// استخدام middleware 'can' في تعريف المسار
    // The current user may update the post...    // ... الكود الذي يتم تنفيذه إذا كان مصرحًا
})->middleware('can:update,post');
```

في هذا المثال، نقوم بتمرير وسيطتين إلى can middleware. الوسيط الأول هو اسم الإجراء الذي نرغب في تفويضه، والثاني هو معلمة المسار التي نرغب في تمريرها إلى طريقة السياسة.

في هذه الحالة، نظرًا لأننا نستخدم ربط النموذج الضمني، فسيتم تمرير نموذج App\Models\Post إلى طريقة السياسة.

إذا لم يكن المستخدم مخولاً بتنفيذ الإجراء المحدد، فسيتم إرجاع استجابة HTTP برمز حالة ٤٠٣ بواسطة البرنامج الوسيط.

لمزيد من السهولة، يمكنك أيضًا إرفاق الوسيط can بمسارك باستخدام الدالة can:

```
use App\Models\Post;
Route::put('/post/{post}', function (Post $post) {// إرفاق middleware طريقة مختصرة لدالة can() التفويض باستخدام
    // The current user may update the post...    // ... الكود الذي يتم تنفيذه إذا كان مصرحًا
})->can('update', 'post');
```

الإجراءات التي لا تتطلب نماذج(Actions That Don't Require Models)

الإجراءات التي لا تتطلب نماذج

مرة أخرى، لا تتطلب بعض دوال الـ policy مثل الإنشاء مثيلًا للنموذج. في هذه المواقف، يمكنك تمرير اسم فئة إلى البرنامج الوسيط. سيتم استخدام اسم الفئة لتحديد السياسة التي سيتم استخدامها عند تفويض الإجراء:

```
Route::post('/post', function () {// تمرير اسم الـ Model class إلى الـ middleware
    // The current user may create posts...    // ... الكود الذي يتم تنفيذه إذا كان مصرحًا
})->middleware('can:create,App\Models\Post');
```

قد يصبح تحديد اسم الفصل بالكامل داخل تعريف الوسيط النصي أمرًا مرهقًا. ولهذا السبب، يمكنك اختيار إرفاق الوسيط النصي can بطريقك باستخدام الدالة can:

```
use App\Models\Post;

Route::post('/post', function () {// طريقة مختصرة لتفويض إجراء إنشاء منشور باستخدام دالة can()
   // The current user may create posts...  // ... الكود الذي يتم تنفيذه إذا كان مصرحًا
})->can('create', Post::class);
```

عبر قوالب Blade (Via Blade Templates)

عند كتابة قوالب Blade، قد ترغب في عرض جزء من الصفحة فقط إذا كان المستخدم مخولاً بتنفيذ إجراء معين. على سبيل المثال، قد ترغب في عرض نموذج تحديث لمنشور مدونة فقط إذا كان المستخدم قادرًا بالفعل على تحديث المنشور. في هذه الحالة، يمكنك استخدام التوجيهات @can و@cannot:

```
@can('update', $post)
   <!-- The current user can update the post... -->    {{-- عرض نموذج التحديث أو الأزرار --}}
@elsecan('create', App\Models\Post::class)
   <!-- The current user can create new posts... -->            {{-- عرض رابط إنشاء منشور جديد --}}
@else
   <!-- ... -->    {{-- محتوى بديل --}}
@endcan

@cannot('update', $post)
   <!-- The current user cannot update the post... -->    {{-- عرض رسالة أن المستخدم لا يمكنه التحديث --}}
@elsecannot('create', App\Models\Post::class)
   <!-- The current user cannot create new posts... -->    {{-- عرض رسالة أن المستخدم لا يمكنه الإنشاء --}}
@endcannot
```

هذه التوجيهات عبارة عن اختصارات ملائمة لكتابة عبارات @if و@unless. عبارات @can و@cannot أعلاه تعادل العبارات التالية:

```
@if (Auth::user()->can('update', $post))
   <!-- The current user can update the post... --> {{-- عرض نموذج التحديث أو الأزرار --}}
@endif

@unless (Auth::user()->can('update', $post))
   <!-- The current user cannot update the post... --> {{-- عرض رسالة أن المستخدم لا يمكنه التحديث --}}
@endunless
```

يمكنك أيضًا تحديد ما إذا كان المستخدم مخولًا بتنفيذ أي إجراء من مجموعة معينة من الإجراءات. لتحقيق ذلك، استخدم التوجيه @canany:

```
@canany(['update', 'view', 'delete'], $post)
   <!-- The current user can update, view, or delete the post... -->    {{-- عرض الأزرار أو الإجراءات المسموح بها --}}
@elsecanany(['create'], \App\Models\Post::class)
   <!-- The current user can create a post... -->    {{-- عرض رابط إنشاء منشور جديد --}}
@endcanany
```

الإجراءات التي لا تتطلب نماذج(Actions That Don't Require Models)

مثل معظم دوال التفويض الأخرى، يمكنك تمرير اسم فئة إلى التوجيهات @can و@cannot إذا كان الإجراء لا يتطلب مثيل نموذج:

```
@can('create', App\Models\Post::class)
   <!-- The current user can create posts... -->    {{-- عرض الأزرار أو الإجراءات المسموح بها --}}
@endcan

@cannot('create', App\Models\Post::class)
   <!-- The current user can't create posts... -->    {{-- عرض رابط إنشاء منشور جديد --}}
@endcannot
```

توفير سياق إضافي(Supplying Additional Context)

عند تفويض الإجراءات باستخدام السياسات، يمكنك تمرير مصفوفة كحجة ثانية إلى وظائف التفويض والمساعدات المختلفة. سيتم استخدام العنصر الأول في المصفوفة لتحديد السياسة التي يجب استدعاؤها، بينما يتم تمرير بقية عناصر المصفوفة كمعلمات إلى طريقة السياسة ويمكن استخدامها لسياق إضافي عند اتخاذ قرارات التفويض. على سبيل المثال، ضع في اعتبارك تعريف طريقة PostPolicy التالي الذي يحتوي على معلمة $category إضافية:

```
/**
 * Determine if the given post can be updated by the user.
 */
public function update(User $user, Post $post, int $category): bool
```

```
{
   return $user->id === $post->user_id &&
       $user->canUpdateCategory($category);
}
```

عند محاولة تحديد ما إذا كان المستخدم المعتمد قادرًا على تحديث منشور معين، يمكننا استدعاء طريقة السياسة هذه على النحو التالي:

```
/**
 * Update the given blog
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function update(Request $request, Post $post): RedirectResponse
{
   Gate::authorize('update', [$post, $request->category]);

   // The current user can update the blog post...
   return redirect('/posts');
}
```

التفويض و الـ Inertia(Authorization & Inertia)

على الرغم من أنه يجب دائمًا التعامل مع التفويض على الخادم، فقد يكون من الملائم في كثير من الأحيان تزويد تطبيق الواجهة الأمامية ببيانات التفويض من أجل عرض واجهة المستخدم الخاصة بتطبيقك بشكل صحيح. لا يحدد Laravel اتفاقية مطلوبة لعرض معلومات التفويض على واجهة أمامية مدعومة بالقصور الذاتي.

ومع ذلك، إذا كنت تستخدم إحدى مجموعات البدء المستندة إلى Inertia في Laravel، فإن تطبيقك يحتوي بالفعل على برنامج وسيط HandleInertiaRequests. داخل طريقة المشاركة الخاصة بهذا البرنامج الوسيط، يمكنك إرجاع البيانات المشتركة التي سيتم توفيرها لجميع صفحات Inertia في تطبيقك. يمكن أن تعمل هذه البيانات المشتركة كموقع مناسب لتحديد معلومات التفويض للمستخدم:

```
<?php

namespace App\Http\Middleware;

use App\Models\Post;
use Illuminate\Http\Request;
use Inertia\Middleware;

class HandleInertiaRequests extends Middleware
{
   /**
    * Define the props that are shared by default.
    *
    * @return array<string, mixed>
    */
   public function share(Request $request)
   {
      return [
         ...parent::share($request),
         'auth' => [
            'user' => $request->user(),
            'permissions' => [
               'post' => [
                  'create' => $request->user()->can('create', Post::class),
               ],
            ],
         ],
      ];
   }
}
```