KAPITEL 5

Datenbanken und Eloquent

Laravel bietet eine Reihe von Tools, um mit den Datenbanken Ihrer Anwendung zu interagieren, aber das bemerkenswerteste dieser Werkzeuge ist Eloquent, Laravels ActiveRecord, der objektrelationale Mapper.

Eloquent ist eines der beliebtesten und wirkungsmächtigsten Features von Laravel und ein hervorragendes Beispiel dafür, inwiefern sich Laravel von den meisten anderen PHP-Frameworks unterscheidet: Während die meisten ORMs zwar leistungsstark, aber sehr komplex sind, zeichnet sich Eloquent durch seine Einfachheit aus. Zu jeder Tabelle existiert eine eigene Klasse, die für das Abrufen, Darstellen und Speichern von Daten aus dieser und in diese Tabelle verantwortlich ist.

Aber auch, wenn Sie Eloquent nicht einsetzen, profitieren Sie von den weiteren Datenbanktools in Laravel. Bevor wir in Eloquent einsteigen, werden wir deshalb zunächst die Grundlagen der Datenbankfunktionalität von Laravel behandeln: Migrationen, Seeding – das Erstellen von Test- bzw. Anfangsdaten – und den Query Builder.

Danach erst werden wir uns mit Eloquent befassen: mit der Definition der Modelle, dem Einfügen, Aktualisieren und Löschen von Daten, wie beim Lesen und Schreiben der Daten Zugriffsmethoden, Mutatoren und Attribut-Casting eingesetzt werden können und schließlich mit den Beziehungen zwischen den Tabellen. Das klingt erst einmal ziemlich komplex, aber wir werden alles Schritt für Schritt und ganz in Ruhe durchgehen.

Konfiguration

Bevor wir uns mit der Verwendung der Datenbanktools von Laravel befassen, lassen Sie uns eine Sekunde innehalten und die Konfiguration Ihrer Datenbank-Anmeldeinformationen und Verbindungen betrachten.

Die Konfiguration für den Datenbankzugriff wird in den beiden Dateien *config/database.php* und *.env* verwaltet. Wie in vielen anderen Konfigurationsbereichen von Laravel können Sie mehrere »Verbindungen« definieren und dann entscheiden, welche davon standardmäßig verwendet werden soll.

Datenbankverbindungen

Standardmäßig gibt es für jeden der Treiber eine Verbindung, wie Sie in Beispiel 5-1 sehen können.

Beispiel 5-1: Standard-Datenbankverbindungen

```
    'connections' => [

        'sqlite' => [
            'driver' => 'sqlite',
            'url' => env('DATABASE_URL'),
            'database' => env('DB_DATABASE', database_path('database.sqlite')),
            'prefix' => '',
            'foreign_key_constraints' => env('DB_FOREIGN_KEYS', true),
        ],

        'mysql' => [
            'driver' => 'mysql',
            'url' => env('DATABASE_URL'),
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'unix_socket' => env('DB_SOCKET', ''),
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'prefix_indexes' => true,
            'strict' => true,
            'engine' => null,
            'options' => extension_loaded('pdo_mysql') ? array_filter([
                PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
                ]) : [],
        ],

        'pgsql' => [
            'driver' => 'pgsql',
            'url' => env('DATABASE_URL'),
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '5432'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'prefix' => '',
            'prefix_indexes' => true,
            'schema' => 'public',
            'sslmode' => 'prefer',
        ],

        'sqlsrv' => [
            'driver' => 'sqlsrv',
            'url' => env('DATABASE_URL'),
            'host' => env('DB_HOST', 'localhost'),
            'port' => env('DB_PORT', '1433'),
            'database' => env('DB_DATABASE', 'forge'),
```

```
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'prefix' => '',
            'prefix_indexes' => true,
        ],

    ]
```

Nichts hindert Sie daran, diese benannten Verbindungen zu löschen, zu ändern oder eigene zu erstellen. Sie können neue benannte Verbindungen erstellen und die dabei zu verwendenden Treiber (MySQL, Postgres usw.) selbst festlegen. Obwohl es standardmäßig nur eine Verbindung pro Treiber gibt, existiert keine diesbezügliche Einschränkung: Sie können beispielsweise auch fünf verschiedene Verbindungen mit dem `mysql`-Treiber anlegen, wenn Sie möchten.

Jede Verbindung ermöglicht es Ihnen, die Eigenschaften zu definieren, die für diesen Verbindungstyp und dessen Anpassung erforderlich sind.

Es gibt einige Gründe dafür, dass mehrere Treiber angelegt sind. Zunächst einmal dient der Abschnitt »connections« als eine einfache Vorlage, damit man schnell Anwendungen mit einem der unterstützten Datenbankverbindungstypen aufsetzen kann. Bei vielen Anwendungen reicht es, die zu verwendende Datenbankverbindung auszuwählen, die benötigten Informationen anzugeben und die anderen Abschnitte zu löschen. Ich selbst behalte normalerweise einfach alle, nur für den Fall, dass ich sie irgendwann doch noch benötige.

Aber es mag auch Fälle geben, in denen Sie mehrere Verbindungen innerhalb derselben Anwendung einsetzen möchten. Beispielsweise könnten Sie verschiedene Datenbankverbindungen für unterschiedliche Arten von Daten verwenden oder aus einer bestimmten Datenbank lesen und in eine andere schreiben. Da man mehrere Verbindungen definieren kann, ist das schnell eingerichtet.

Weitere Optionen zur Konfiguration von Datenbanken

Die Konfigurationsdatei *config/database.php* enthält eine Reihe weiterer Einstellungen. Sie können u. a. den Zugriff auf Redis konfigurieren, den für Migrationen verwendeten Tabellennamen anpassen und die Standard-Datenbankverbindung bestimmen.

Für jeden Dienst in Laravel, der Verbindungen aus mehreren Quellen zulässt – beispielsweise können Sessions in einer Datenbank oder im Dateisystem gesichert werden, der Cache kann Redis oder Memcached nutzen, es können MySQL- oder PostgreSQL-Datenbanken verwendet werden –, können Sie mehrere solcher Verbindungen definieren und auch festlegen, dass eine bestimmte Verbindung als Default dienen, also immer dann verwendet werden soll, wenn nicht explizit eine andere Verbindung vorgegeben wird. So legen Sie bei Bedarf eine bestimmte Verbindung fest:

```
$users = DB::connection('secondary')->select('select * from users');
```

Migrationen

In modernen Frameworks wie Laravel kann man die Datenbankstruktur sehr einfach über codegesteuerte Migrationen festlegen. Jede neue Tabelle, Spalte, jeder Index und Schlüssel wird im Code definiert, sodass in einer neuen Umgebung in Sekundenschnelle das gesamte Datenbankschema einer Anwendung repliziert werden kann.

Migrationen definieren

Eine Migration ist eine einzelne Datei, in der zwei Dinge festgelegt werden: in welcher Weise die Datenbank beim Ausführen dieser Migration in der sogenannten *up*-Richtung geändert werden soll und – optional – die gewünschten Änderungen beim Ausführen dieser Migration in der entgegengesetzten *down*-Richtung.

»Up« und »Down« bei Migrationen

Migrationen werden immer in chronologischer Reihenfolge ausgeführt. Jede Migrationsdatei wird nach diesem Muster benannt: *2020_01_12_000000_create_users_table.php*. Wenn ein neues System migriert wird, wird die `up()`-Methode jeder Migration, beginnend mit dem frühesten Datum, ausgeführt – Sie migrieren an dieser Stelle »up«, also »aufwärts« oder »vorwärts«. Das Migrationssystem ermöglicht es Ihnen aber auch, Ihre letzten Migrationen zurückzusetzen. Es führt dazu die `down()`-Methoden der entsprechenden Migrationen aus, die alle Änderungen der `up()`-Methoden rückgängig machen sollen.

Die `up()`-Methode einer Migration führt also die gewünschten Änderungen durch, und die `down()`- Methode macht sie wieder rückgängig.

Beispiel 5-2 zeigt, wie die standardmäßige »create_users_table«-Migration aussieht, die mit Laravel ausgeliefert wird.

Beispiel 5-2: Laravels Standard-Migration »create_users_table«

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
```

```
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

E-Mail-Verifizierung

Die Spalte `email_verified_at` ist nur in Anwendungen vorhanden, die in Laravel 5.7 und höher entwickelt wurden. In ihr wird ein Zeitstempel gespeichert, der angibt, wann ein registrierter Benutzer seine E-Mail-Adresse bestätigt hat.

Wie Sie sehen können, gibt es eine `up()`- und eine `down()`-Methode. `up()` weist die Migration an, eine neue Tabelle namens `users` mit einigen Feldern zu erstellen, und `down()` sagt ihr, dass sie die Tabelle `users` wieder löschen soll.

Erstellen einer Migration

Wie Sie in Kapitel 8 sehen werden, stellt Laravel eine Reihe von Befehlszeilen-Tools zur Verfügung, mit denen Sie mit Ihrer Applikation interagieren und teilweise vorausgefüllte Dateien erzeugen können. Mit einem dieser Befehle können Sie eine Migrationsdatei erstellen. Dazu führen Sie den Befehl `php artisan make:migration` aus, der nur einen einzigen Parameter benötigt: den Namen der Migration. Um zum Beispiel die Tabellenmigration zu erstellen, die wir uns gerade angeschaut haben, würden Sie `php artisan make:migration create_users_table` eingeben (allerdings hätten Sie dann nur ein Skelett der Migrationsdatei, denn die eigentlichen Spaltendefinitionen würden fehlen).

Es gibt zwei Flags, die Sie optional an diesen Befehl übergeben können. `--create=`*`table_name`* füllt die Migration mit dem nötigen Code, um eine Tabelle mit dem Namen *`table_name`* zu erstellen, und `--table=`*`table_name`* erstellt eine Migration, die die nötigen Befehle enthält, um Änderungen an einer bereits bestehenden Tabelle vorzunehmen. Hier sind einige Beispiele:

```
php artisan make:migration create_users_table
php artisan make:migration add_votes_to_users_table --table=users
php artisan make:migration create_users_table --create=users
```

Tabellen anlegen

Wir haben bereits in der mitgelieferten `create_users_table`-Migration gesehen, dass unsere Migrationen von der `Schema`-Fassade und deren Methoden abhängen. Alles, was wir in diesen Migrationen durchführen können, stützt sich auf die Methoden von `Schema`.

Um in einer Migration eine neue Tabelle zu erstellen, verwenden Sie die Methode `create()` – der erste Parameter ist der Tabellenname und der zweite eine Closure, also eine anonyme Funktion, in der die Spalten definiert werden:

```
Schema::create('users', function (Blueprint $table) {
    // Hier werden die Spalten definiert
});
```

Spalten anlegen

Um neue Spalten in einer Tabelle anzulegen, sei es bei der Erstellung oder der nachträglichen Änderung einer Tabelle, verwenden Sie die Instanz von `Blueprint`, die an die Closure-Funktion übergeben wird:

```
Schema::create('users', function (Blueprint $table) {
    $table->string('name');
});
```

Betrachten wir die verschiedenen Methoden, die den `Blueprint`-Instanzen zum Erstellen von Spalten zur Verfügung stehen. Ich beschreibe hier, wie es in MySQL funktioniert, aber falls Sie eine andere Datenbank einsetzen, verwendet Laravel ggf. einfach das nächstbeste Äquivalent.

Im Folgenden finden Sie die `Blueprint`-Methoden, um einfache Spalten anzulegen:

`integer(`*`colName`*`)`, `tinyInteger(`*`colName`*`)`, `smallInteger(`*`colName`*`)`, `mediumInteger(`*`colName`*`)`, `bigInteger(`*`colName`*`)`
: Fügt eine Spalte vom Typ `INTEGER` oder eine ihrer vielen Varianten hinzu

`string(`*`colName, length`*`)`
: Fügt eine Spalte vom Typ `VARCHAR` mit einer optionalen Länge hinzu

`binary(`*`colName`*`)`
: Fügt eine Spalte vom Typ `BLOB` hinzu

`boolean(`*`colName`*`)`
: Fügt eine Spalte vom Typ `BOOLEAN` hinzu (ein `TINYINT(1)` in MySQL)

`char(`*`colName, length`*`)`
: Fügt eine Spalte vom Typ `CHAR` mit einer optionalen Länge hinzu

`datetime(`*`colName`*`)`
: Fügt eine Spalte vom Typ `DATETIME` hinzu

`decimal(colName, precision, scale)`
: Fügt eine `DECIMAL`-Spalte mit einer bestimmten Genauigkeit (maximale Anzahl der Stellen) und Anzahl der Nachkommastellen hinzu – z.B. legt `decimal('amount', 5, 2)` eine Genauigkeit von 5 mit 2 Nachkommastellen fest

`double(colName, total digits, digits after decimal)`
: Fügt eine `DOUBLE`-Spalte hinzu – `double('tolerance', 12, 8)` legt beispielsweise 12 Stellen fest, wobei 8 dieser Stellen rechts vom Dezimalpunkt liegen wie in `7204.05691739`

`enum(colName, [choiceOne, choiceTwo])`
: Fügt eine Spalte vom Typ `ENUM` hinzu, mit den vorgegebenen Auswahlmöglichkeiten

`float(colName, precision, scale)`
: Fügt eine Spalte vom Typ `FLOAT` (wie `DOUBLE` in MySQL) hinzu

`json(colName)` *und* `jsonb(colName)`
: Fügt eine `JSON`- oder `JSONB`-Spalte hinzu (oder eine `TEXT`-Spalte in Laravel 5.1)

`text(colName)`, `mediumText(colName)`, `longText(colName)`
: Fügt eine `TEXT`-Spalte (oder deren verschiedene Größen) hinzu

`time(colName)`
: Fügt eine Spalte vom Typ `TIME` hinzu

`timestamp(colName)`
: Fügt eine `TIMESTAMP`-Spalte hinzu

`uuid(colName)`
: Fügt eine Spalte vom Typ `UUID` hinzu (`CHAR(36)` in MySQL)

Und das sind die komplexeren `Blueprint`-Methoden:

`increments(colName)` *und* `bigIncrements(colName)`
: Fügt eine inkrementelle `INTEGER`- oder `BIG INTEGER`-Primärschlüssel-ID ohne Vorzeichen hinzu

`timestamps()` *und* `nullableTimestamps()`
: Fügt `created_at`- und `updated_at`-Zeitstempel-Spalten hinzu

`rememberToken()`
: Fügt eine `remember_token`-Spalte (`VARCHAR(100)`) für »remember me«-Benutzer-Tokens hinzu

`softDeletes()`
: Fügt einen `deleted_at`-Zeitstempel für die Verwendung mit Soft-Deletes hinzu

`morphs(colName)`
: Fügt eine Integer-Spalte `colName_id` und eine Zeichenketten-Spalte `colName_type` zur Verwendung in polymorphen Beziehungen hinzu – z.B. würde `morphs(eigenschaft)` die Integer-Spalte `eigenschaft_id` und die Zeichenketten-Spalte `eigenschaft_type` ergänzen

Zusätzliche Eigenschaften verkettet definieren

Die meisten Eigenschaften eines Felds – seine Länge zum Beispiel – werden als zweiter Parameter der Felderstellungsmethode festgelegt, wie wir im vorherigen Abschnitt gesehen haben. Aber es gibt noch ein paar andere Eigenschaften, die durch weitere Methodenaufrufe direkt nach der Erstellung der Spalte angehängt werden können. Beispielsweise soll hier die Spalte `email` Nullwerte enthalten dürfen und (in MySQL) direkt nach der Spalte `last_name` platziert werden:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable()->after('last_name');
});
```

Die folgenden Methoden werden verwendet, um zusätzliche Eigenschaften eines Felds festzulegen:

`nullable()`
: Erlaubt die Verwendung von `NULL`-Werten in einer Spalte

`default('default content')`
: Legt den Standardinhalt für diese Spalte fest, für den Fall, dass kein anderer Wert spezifiziert wird

`unsigned()`
: Legt fest, dass die Ganzzahl-Werte einer Integerspalte kein Vorzeichen haben dürfen

`first()` *(nur MySQL)*
: Platziert eine Spalte ganz am Anfang einer Tabelle

`after(colName)` *(nur MySQL)*
: Platziert eine Spalte hinter einer anderen Spalte

`unique()`
: Fügt einen eindeutigen Index hinzu

`primary()`
: Fügt einen Primärschlüsselindex hinzu

`index()`
: Fügt einen Basisindex hinzu

Bitte beachten Sie, dass die Methoden `unique()`, `primary()` und `index()` auch an anderer Stelle als in »flüssigen«, verketteten Spaltendefinitionen verwendet werden können – darauf kommen wir später zurück.

Tabellen löschen

Um eine Tabelle zu löschen, können Sie die `dropIfExists()`-Methode der `Schema`-Fassade nutzen, die den Tabellennamen als Parameter erwartet:

```
Schema::dropIfExists('contacts');
```

Spalten ändern

Um eine Spalte zu modifizieren, reicht es, an den gleichen Code, den man benutzt, um diese Spalte erstmals zu erstellen, einen Aufruf der Methode `change()` anzuhängen.

Eine Abhängigkeit, die vor dem Ändern von Spalten installiert werden muss

Bevor Sie Spalten ändern können (und auch bevor Sie mit SQLite Spalten löschen können), müssen Sie `composer require doctrine/dbal` ausführen, um das entsprechende Paket zu installieren.

Wenn wir beispielsweise eine Zeichenkettenspalte `name` haben, deren Länge von `255` auf `100` geändert werden soll, würde der entsprechende Befehl so aussehen:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 100)->change();
});
```

Das Gleiche gilt, wenn wir eine Eigenschaft anpassen wollen, dies aber nicht – anders als bei `string('name', 100)` – im Methodenaufruf selbst möglich ist. Um für eine Spalte Nullwerte zuzulassen, wird beispielsweise `nullable()` vor `change()` eingefügt:

```
Schema::table('contacts', function (Blueprint $table)
{
    $table->string('deleted_at')->nullable()->change();
});
```

So benennen wir eine Spalte um:

```
Schema::table('contacts', function (Blueprint $table)
{
    $table->renameColumn('promoted', 'is_promoted');
});
```

Und so wird eine Spalte gelöscht:

```
Schema::table('contacts', function (Blueprint $table)
{
    $table->dropColumn('votes');
});
```

Mehrere Spalten in SQLite gleichzeitig ändern

Bei SQLite führt es zu einer Fehlermeldung, wenn man innerhalb einer Migrations-Closure mehrere Spalten löschen oder ändern will.

In Kapitel 12 empfehle ich, dass Sie SQLite als Testdatenbank verwenden, selbst wenn Sie ansonsten eine traditionellere Datenbank einsetzen – sollten Sie diesem Ratschlag folgen und beim Testen SQLite verwenden, sollten Sie diese Einschränkung sicherheitshalber grundsätzlich beachten.

Sie müssen deshalb jedoch nicht gleich mehrere Migrationen anlegen. Stattdessen reicht es, `Schema::table()` innerhalb der `up()`-Methode Ihrer Migration mehrfach aufzurufen:

```
public function up()
{
    Schema::table('contacts', function (Blueprint $table)
```

```
        {
            $table->dropColumn('is_promoted');
        });

        Schema::table('contacts', function (Blueprint $table)
        {
            $table->dropColumn('alternate_email');
        });
    }
```

Indizes und Fremdschlüssel

Wir haben bisher gesehen, wie man Spalten erstellt, ändert und löscht. Wenden wir uns jetzt der Erstellung von Indizes und deren Verknüpfung zu.

Falls Sie nicht mit Indizes vertraut sind: Ihre Datenbanken funktionieren zwar auch, wenn Sie keinerlei Indizes verwenden, aber sie sind ausgesprochen wichtig für die Leistungsoptimierung und für die Gewährleistung der Datenintegrität von verknüpften Tabellen. Deshalb würde ich empfehlen, hier weiterzulesen und den folgenden Abschnitt nur dann (vorerst) zu überspringen, wenn es unbedingt sein muss.

Indizes hinzufügen. In Beispiel 5-3 finden Sie Beispiele, wie Sie eine Spalte indizieren können.

Beispiel 5-3: Hinzufügen von Indizes in Migrationen

```
// Nachdem die Spalten erstellt wurden ...
$table->primary('primary_id'); // Primärschlüssel; unnötig, wenn eine
                               // autoinkrementelle Spalte (meistens +id+) verwendet wird
$table->primary(['first_name', 'last_name']); // Zusammengesetzter Schlüssel
$table->unique('email'); // Eindeutiger Index
$table->unique('email', 'optional_custom_index_name'); // Eindeutiger Index mit
                                                       // Namensvorgabe
$table->index('amount'); // Basisindex
$table->index('amount', 'optional_custom_index_name'); // Basisindex mit Namensvorgabe
```

Beachten Sie, dass das erste Beispiel – `primary('primary_id')` – nicht erforderlich ist, wenn Sie bei der Spaltendefinition die `increments()`- oder `bigIncrements()`-Methode benutzen; dadurch wird automatisch ein Primärschlüsselindex hinzugefügt.

Indizes löschen. Wie Indizes entfernt werden können, zeigt Beispiel 5-4.

Beispiel 5-4: Entfernen von Indizes in Migrationen

```
$table->dropPrimary('contacts_id_primary');
$table->dropUnique('contacts_email_unique');
$table->dropIndex('optional_custom_index_name');

// Wenn Sie ein Array von Spaltennamen an dropIndex übergeben, wird die Methode
// versuchen, die Indexnamen basierend auf Laravels Generierungskonventionen zu erraten
$table->dropIndex(['email', 'amount']);
```

Hinzufügen und Entfernen von Fremdschlüsseln. Um einen Fremdschlüssel hinzuzufügen, der definiert, dass eine bestimmte Spalte auf eine Spalte in einer anderen Tabelle verweist, bietet Laravel eine einfache und »sprechende« Syntax:

```
$table->foreign('user_id')->references('id')->on('users');
```

Hier fügen wir einer Spalte `user_id` einen Fremdschlüsselindex hinzu, der auf die Spalte `id` in der Tabelle `users` verweist. Einfacher geht es nicht.

Wenn wir Fremdschlüsselbeschränkungen angeben wollen, können wir das mit `onDelete()` oder `onUpdate()` tun. Zum Beispiel:

```
$table->foreign('user_id')
    ->references('id')
    ->on('users')
    ->onDelete('cascade');
```

Um einen Fremdschlüssel zu löschen, können wir entweder auf seinen Indexnamen verweisen (der automatisch aus den Namen der indizierten Tabellen und Spalten gebildet wird) ...

```
$table->dropForeign('contacts_user_id_foreign');
```

... oder wir übergeben ein Array der Felder, auf die sich der Schlüssel in der lokalen Tabelle bezieht:

```
$table->dropForeign(['user_id']);
```

Migrationen ausführen

Nachdem Sie Ihre Migrationen definiert haben, möchten Sie vielleicht wissen, wie Sie sie ausführen können. Dafür gibt es einen Artisan-Befehl:

```
php artisan migrate
```

Dieser Befehl wendet alle »offenen« Migrationen an (indem jeweils deren `up()`-Methode ausgeführt wird). Laravel zeichnet auf, welche Migrationen Sie bereits durchgeführt haben und welche nicht. Dadurch kann dieser Befehl überprüfen, ob alle verfügbaren Migrationen bereits ausgeführt wurden – falls nicht, wird das nachgeholt.

Bei diesem Befehl können Sie verschiedene Optionen angeben. Erstens können Sie Migrationen *und* Seeding (das wir als Nächstes behandeln werden) auf einen Schlag durchführen:

```
php artisan migrate --seed
```

Außerdem können Sie auch eine der folgenden Befehlsvarianten nutzen:

`migrate:install`
: Erstellt die Datenbanktabelle, in der aufgezeichnet wird, welche Migrationen es gibt; diese Tabelle wird aber bei Bedarf automatisch angelegt, wenn Sie erstmals Migrationen durchführen, sodass Sie diesen separaten Befehl normalerweise nicht benötigen.

`migrate:reset`
: Führt ein Rollback aller Datenbankmigrationen durch, die Sie auf dieser Instanz der Anwendung ausgeführt haben.

`migrate:refresh`
: Führt zuerst ein Rollback aller Datenbankmigrationen durch, die Sie auf dieser Instanz der Anwendung ausgeführt haben, und wendet dann jede verfügbare Migration erneut an. Damit ist es eine Kombination von `migrate:reset` und einem vollständigen `migrate` – beides wird nacheinander ausgeführt.

`migrate:fresh`
: Löscht alle Tabellen und führt alle Migrationen erneut durch. Das ist praktisch dasselbe wie `refresh`, hält sich aber nicht mit den »down«-Methoden der Migrationen auf – alle Tabellen werden vollständig gelöscht und dann erneut alle »up«-Methoden ausgeführt.

`migrate:rollback`
: Führt nur ein Rollback der Migrationen aus, die beim letzten Aufruf des `migrate`-Befehls stattgefunden haben, oder »rollt« – mit der zusätzlichen Option `--step=`*n* – die angegebene Anzahl von Migrationen zurück.

`migrate:status`
: Zeigt eine Tabelle mit allen Migrationen an, wobei jeweils ein `Y` oder `N` anzeigt, ob sie in der aktuellen Umgebung bereits ausgeführt wurden oder nicht.

Migrationen mit Homestead/Vagrant

Wenn Sie Migrationen auf Ihrem lokalen Rechner ausführen und Ihre *.env*-Datei auf eine Datenbank in einer Vagrant-Box verweist, werden Ihre Migrationen fehlschlagen. Sie müssen dann mit `ssh` auf die Vagrant-Box zugreifen und die Migrationen dort ausführen. Dasselbe gilt für Seeds und alle anderen Artisan-Befehle, die die Datenbank ändern oder aus ihr lesen.

Seeding

Das Seeding mit Laravel ist derart einfach, dass es sich als Teil des normalen Entwicklungsablaufs in einer Weise durchgesetzt hat, die es in früheren PHP-Frameworks noch nicht gegeben hat. Es gibt einen Ordner *database/seeds*, der die Klasse `DatabaseSeeder` enthält, deren `run()`-Methode aufgerufen wird, wenn Sie ein Seeding durchführen.

Es gibt zwei Varianten, ein Seeding zu starten: zusammen mit einer Migration oder separat.

Um das Seeding zusammen mit Migrationen auszuführen, fügen Sie einfach einem Migrationsaufruf `--seed` hinzu:

```
php artisan migrate --seed
php artisan migrate:refresh --seed
```

Und um es separat auszuführen:

```
php artisan db:seed
php artisan db:seed --class=VotesTableSeeder
```

Dies ruft standardmäßig die Methode `run()` der Klasse `DatabaseSeeder` oder der durch `--class` angegebenen Seeder-Klasse auf.

Eine Seeder-Klasse anlegen

Um einen Seeder zu erstellen, verwenden Sie den Artisan-Befehl `make:seeder`:

```
php artisan make:seeder ContactsTableSeeder
```

Damit wird eine Klasse `ContactsTableSeeder` im Verzeichnis *database/seeds* erstellt. Bevor wir diese bearbeiten, fügen wir der Klasse `DatabaseSeeder` einen entsprechenden Aufruf hinzu, wie in Beispiel 5-5 gezeigt, damit die neue Klasse später automatisch mit aufgerufen wird.

Beispiel 5-5: Aufruf einer benutzerdefinierten Seeder-Klasse aus DatabaseSeeder.php

```
// database/seeds/DatabaseSeeder.php
...
    public function run()
    {
        $this->call(ContactsTableSeeder::class);
    }
```

Lassen Sie uns nun die Seeder-Klasse bearbeiten. Die einfachste Variante besteht darin, manuell einen Datensatz mithilfe der `DB`-Fassade einzufügen, wie in Beispiel 5-6 veranschaulicht.

Beispiel 5-6: Datenbankeinträge mit einer benutzerdefinierten Seeder-Klasse anlegen

```
<?php

use Illuminate\Database\Seeder;

class ContactsTableSeeder extends Seeder
{
    public function run()
    {
        DB::table('contacts')->insert([
            'name' => 'Lupita Smith',
            'email' => 'lupita@gmail.com',
        ]);
    }
}
```

So erhalten wir einen einzelnen Datensatz, was ein guter Anfang ist. Aber für wirklich brauchbare Seeds wäre es wahrscheinlich sinnvoller, eine Schleife über irgendeine Art von Zufallsgenerator laufen zu lassen und damit `insert()` mehrmals auszuführen, oder? Genau solch ein Feature bietet Laravel.

Modellfabriken

Modellfabriken (engl. *model factories*) definieren Muster zur Erstellung von Testeinträgen – also »gefakten« Daten – für Ihre Datenbanktabellen. Standardmäßig ist jede Fabrik nach einer Eloquent-Klasse benannt, aber Sie können sie auch einfach nach der Tabelle benennen, wenn Sie nicht mit Eloquent arbeiten wollen. Beispiel 5-7 zeigt die beiden Varianten, eine Factory zu definieren:

Beispiel 5-7: Definition von Modellfabriken per Eloquent-Klasse bzw. Tabellenname

```
$factory->define(User::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
    ];
});

$factory->define('users', function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
    ];
});
```

Theoretisch können Sie diese Fabriken beliebig benennen, aber sich nach der Eloquent-Klasse zu richten, ist der gebräuchlichste Weg.

Erstellen einer Modellfabrik

Modellfabriken befinden sich in *database/factories*. Ab Laravel 5.5 ist jede Fabrik in der Regel in ihrer eigenen Klasse definiert, mit einem Namen und einer Closure, die definiert, wie eine neue Instanz der Klasse erstellt wird. Die Methode `$factory->define()` erwartet den Namen der Factory als ersten Parameter und eine bei jeder Generierung auszuführende Closure als zweiten Parameter.

Modellfabriken vor Laravel 5.5
In Laravel-Versionen vor 5.5 müssen alle Fabriken in der Datei *database/factories/ModelFactory.php* definiert werden. Separate Klassen für die einzelnen Factories gab es vor Version 5.5 noch nicht.

Um eine neue Factory-Klasse zu erzeugen, verwenden Sie den Artisan-Befehl `make:factory`; es ist üblich, auch die Fabrikklassen nach den Eloquent-Modellen zu benennen, für die Instanzen erzeugt werden sollen:

```
php artisan make:factory ContactFactory
```

Dadurch wird eine neue Datei namens *ContactFactory.php* im Verzeichnis *database/factories* erzeugt. Die einfachste Factory für einen einzelnen »Kontakt« könnte etwa so aussehen wie in Beispiel 5-8:

Beispiel 5-8: Einfache Modellfabrik

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
```

```
        'name' => 'Lupita Smith',
        'email' => 'lupita@gmail.com',
    ];
});
```

Jetzt können wir den globalen Helfer `factory()` verwenden, um Instanzen von `Contact` für unser Seeding und Testing zu erstellen:

```
    // Einen Kontakt erzeugen
    $contact = factory(Contact::class)->create();

    // Mehrere Kontakte erzeugen
    factory(Contact::class, 20)->create();
```

Würden wir die Fabrik in dieser Form tatsächlich nutzen, um 20 Kontakte zu erstellen, enthielten alle 20 die gleichen Informationen. Das ist nicht besonders hilfreich.

Richtig ausnutzen können wir Modellfabriken, indem wir der Closure eine Instanz von `Faker` (*https://bit.ly/2FtyJRr*) übergeben; `Faker` erleichtert es, strukturierte Zufallsdaten zu erzeugen. Passen wir also die Fabrik in Beispiel 5-9 entsprechend an.

Beispiel 5-9: Einfache Modellfabrik, modifiziert für die Verwendung von Faker

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
    ];
});
```

Jetzt werden jedes Mal, wenn wir mit dieser Modellfabrik einen gefakten Kontakt erstellen, alle Eigenschaften zufällig generiert.

Eindeutige Daten zufallsbasiert generieren

Wenn Sie sicherstellen möchten, dass die zufällig generierten Werte für ein bestimmtes Feld eindeutig sind, können Sie die `unique()`-Methode von Faker verwenden:

```
return ['email' => $faker->unique()->email];
```

Modellfabriken einsetzen

Es gibt zwei Situationen, in denen man Modellfabriken verwendet: Tests, die wir in Kapitel 12 besprechen werden, und Seeding, um das wir uns hier kümmern. Lassen Sie uns also einen Seeder anlegen, der eine Modellfabrik benutzt; sehen Sie sich dazu bitte Beispiel 5-10 an.

Beispiel 5-10: Modellfabriken einsetzen

```
$post = factory(Post::class)->create([
    'title' => 'My greatest post ever',
]);
```

```
// "Profi"-Fabrik; aber lassen Sie sich nicht abschrecken!
factory(User::class, 20)->create()->each(function ($u) use ($post) {
    $post->comments()->save(factory(Comment::class)->make([
        'user_id' => $u->id,
    ]));
});
```

Um ein Objekt zu erstellen, verwenden wir den globalen Helfer `factory()` und teilen ihm den Namen der Fabrik mit – was, wie wir gerade gesehen haben, gleichzeitig der Name der Eloquent-Klasse ist, von der wir eine Instanz erzeugen. Der Aufruf von `factory()` gibt uns die Fabrik zurück, und dann können wir eine von zwei Methoden darauf anwenden: `make()` oder `create()`.

Beide Methoden erzeugen eine Instanz des angegebenen Modells, unter Verwendung der Definition in der Factory-Datei. Der Unterschied besteht darin, dass `make()` die Instanz erstellt, sie aber (noch) nicht in der Datenbank speichert, während `create()` die Instanz auch sofort in die Datenbank schreibt. In Beispiel 5-10 wurden im zweiten Beispiel beide Methoden verwendet.

Dieses zweite Beispiel – die »Profi«-Fabrik – wird klarer werden, sobald wir im weiteren Verlauf dieses Kapitels die verschiedenen Beziehungen zwischen Eloquent-Modellen behandeln.

Eigenschaften überschreiben beim Aufruf einer Modellfabrik. Wenn man an `make()` oder `create()` ein Array übergibt, kann man bestimmte Eigenschaften der Fabrik überschreiben, wie wir es beispielsweise in Beispiel 5-10 getan haben, um einem Kommentar die `user_id` des Autors zuzuweisen oder den Titel unseres Beitrags manuell festzulegen.

Mit einer Modellfabrik mehr als eine Instanz erzeugen. Wenn Sie dem Helper `factory()` eine Zahl als zweiten Parameter übergeben, können Sie festlegen, dass Sie mehr als eine Instanz erstellen möchten. Anstatt eine einzelne Instanz zurückzugeben, wird nun eine Collection von Instanzen zurückgegeben. Das bedeutet, dass Sie das Ergebnis wie ein Array behandeln, jede seiner Instanzen mit einer anderen Entität verknüpfen oder auf jede Instanz andere Entitätsmethoden anwenden können, wie wir es mit `each()` in Beispiel 5-10 gemacht haben, um für jeden neu erstellten Benutzer auch gleich einen ersten Kommentar zu erzeugen und hinzuzufügen.

»Profi«-Modellfabriken

Nachdem wir uns nun die häufigsten Verwendungsmöglichkeiten von Modellfabriken angeschaut haben, wenden wir uns einigen komplizierteren Einsatzmöglichkeiten zu.

Anhängen von Beziehungen bei der Definition von Modellfabriken. Manchmal muss man zusammen mit einem Hauptelement auch gleich weitere verknüpfte Elemente er-

stellen. Man kann eine Closure verwenden, um ein solches verknüpftes Element zu erstellen und sofort dessen ID zu erhalten, wie in Beispiel 5-11 gezeigt.

Beispiel 5-11: Erstellen eines verknüpften Elements

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => 'Lupita Smith',
        'email' => 'lupita@gmail.com',
        'company_id' => function () {
            return factory(App\Company::class)->create()->id;
        },
    ];
});
```

Die Closure gibt als Wert die Array-Form des erzeugten Elements zurück, so wie sie zu diesem Zeitpunkt vorliegt. Dabei kann man auch auf Werte zurückgreifen, die gerade erst für eine andere Eigenschaft erzeugt wurden, wie Beispiel 5-12 zeigt.

Beispiel 5-12: In einem Seeder Werte aus anderen Eigenschaften verwenden

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => 'Lupita Smith',
        'email' => 'lupita@gmail.com',
        'company_id' => function () {
            return factory(App\Company::class)->create()->id;
        },
        'company_size' => function ($contact) {
            // Nutzt die Eigenschaft "company_id", deren Wert zuvor erzeugt wurde
            return App\Company::find($contact['company_id'])->size;
        },
    ];
});
```

Definition und Zugriff auf mehrere Modellfabrik-Zustände. Gehen wir für eine Sekunde zurück zu *ContactFactory.php* (aus Beispiel 5-8 und Beispiel 5-9). Wir hatten eine einfache Fabrik für Kontakte definiert:

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
    ];
});
```

Aber manchmal braucht man mehr als eine einzige Fabrik für eine Klasse von Objekten. Vielleicht möchten wir Kontakte hinzuzufügen, die als VIPs markiert sein sollen? In diesem Fall kann mit der `state()`-Methode ein weiterer Fabrik-Zustand (engl. *factory state*) definiert werden, wie Beispiel 5-13 zeigt. Der erste Parameter von `state()` ist dabei wiederum der Klassenname, der zweite der Name des neuen Zustands und der dritte ein Array aller Attribute, die Sie speziell für diesen Zustand festlegen möchten.

Beispiel 5-13: Definition mehrerer Fabrik-Zustände für das gleiche Modell

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
    ];
});

$factory->state(Contact::class, 'vip', [
    'vip' => true,
]);
```

Falls die geänderten Attribute mehr als einen einfachen statischen Wert erfordern, können Sie als dritten Parameter eine Closure übergeben, die ein Array der gewünschten Attribute zurückliefert, wie in Beispiel 5-14 zu sehen.

Beispiel 5-14: Definition eines Fabrik-Zustands mit einer Closure

```
$factory->state(Contact::class, 'vip', function (Faker\Generator $faker) {
    return [
        'vip' => true,
        'company' => $faker->company,
    ];
});
```

Lassen Sie uns nun eine Instanz eines bestimmten Zustands erstellen:

```
$vip = factory(Contact::class)->state('vip')->create();

$vips = factory(Contact::class, 3)->state('vip')->create();
```

Fabrik-Zustände vor Laravel 5.3

In Laravel-Versionen vor 5.3 wurden Fabrik-Zustände als Fabrik-Typen (engl. *factory type*) bezeichnet und mit `$factory->defineAs()` anstelle von `$factory->state()` verwendet. Mehr dazu erfahren Sie in der Dokumentation zu Version 5.2 (*https://bit.ly/2Fmnaew*).

Wow. Das war jetzt ganz schön viel Stoff, und machen Sie sich keine Sorgen, falls es etwas anstrengend war, all dem zu folgen, denn der letzte Teil war wirklich ausgesprochen anspruchsvoll. Kommen wir zurück zu den Grundlagen und wenden wir uns dem Kern von Laravels Datenbank-Werkzeugen zu: dem Query Builder bzw. Abfrage-Generator.

Der Query Builder

Jetzt, da Ihre Anwendung mit der Datenbank verbunden ist und Ihre Tabellen migriert und mit Testdaten gefüllt sind, können wir mit der Verwendung der Datenbanktools beginnen. Das Herzstück der Datenbank-Funktionalität von Laravel ist der Query Builder, ein Fluent Interface, mit dem man über eine einzige, konsistente API mit verschiedenen Arten von Datenbanken interagieren kann.

Was ist ein Fluent Interface?

Eine »fließende« oder besser »sprechende« Schnittstelle verwendet in erster Linie die Verkettung von Methoden, um dem Benutzer eine einfachere, besser lesbare API bereitzustellen. Anstatt alle relevanten Daten auf einen Schlag an einen Konstruktor oder eine Methode zu übergeben, können fließende Aufrufketten schrittweise zusammengesetzt werden. Betrachten Sie den folgenden Vergleich:

```
// Nicht fließend: Alles wird als ein langer Ausdruck innerhalb der Klammer
// übergeben
$users = DB::select(['table' => 'users', 'where' => ['type' => 'donor']]);

// Fließend: Drei Methoden werden verkettet, also nacheinander aufgerufen
$users = DB::table('users')->where('type', 'donor')->get();
```

Laravel kann über ein und dieselbe Schnittstelle Verbindungen mit MySQL-, Postgres-, SQLite- und SQL-Server-Datenbanken herstellen – dazu müssen nur einige wenige Konfigurationseinstellungen geändert werden.

Wenn Sie bereits mit anderen PHP-Frameworks gearbeitet haben, kennen Sie wahrscheinlich Werkzeuge, mit denen man Abfragen in »purem« SQL ausführen kann (aus Sicherheitsgründen mit grundlegendem Escaping). So ein Werkzeug ist auch der Query Builder, allerdings mit vielen zusätzlichen Komfortschichten und Helfern. Beginnen wir also mit ein paar einfachen Aufrufen.

Grundlegender Einsatz der DB-Fassade

Bevor wir uns mit der Erstellung komplexer Abfragen mit fließender Methodenverkettung befassen, lassen Sie uns einen Blick auf einige Beispiele für Query-Builder-Befehle werfen. Die `DB`-Fassade wird sowohl für die Verkettung mit dem Query Builder als auch für einfachere Abfragen verwendet, die direktes SQL benutzen, wie in Beispiel 5-15 dargestellt.

Beispiel 5-15: Verwendungsvergleich von direktem SQL und Query Builder

```
// Einfacher SQL-Befehl
DB::statement('drop table users');

// Einfaches SELECT mit Parameterbindung
DB::select('select * from contacts where validated = ?', [true]);

// Abfrage mit fließender Verkettung
$users = DB::table('users')->get();

// Joins und andere komplexe Abfragen
DB::table('users')
    ->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')
             ->where('contacts.type', 'donor');
```

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

Direktes SQL

Wie Sie in Beispiel 5-15 gesehen haben, kann man direkte SQL-Abfragen der Datenbank mithilfe der DB-Fassade und der Methode `statement()` durchführen: `DB::statement('SQL-Befehl hier')`.

Aber es gibt auch spezifische Methoden für verschiedene, häufig benötigte Aktionen: `select()`, `insert()`, `update()` und `delete()`. Das sind zwar ebenfalls direkte Aufrufe, aber mit ein paar kleinen Unterschieden. Erstens liefern `update()` und `delete()` die Anzahl der betroffenen Zeilen zurück, während `statement()` dies nicht tut; zudem ist es für zukünftige Entwickler klarer ersichtlich, welche Art von Befehl ausgeführt wird, wenn Sie diese spezifischen Methoden benutzen.

Direkte SELECTs

Die einfachste der spezifischen DB-Methoden ist `select()`. Sie können es ohne zusätzliche Parameter ausführen:

```
$users = DB::select('select * from users');
```

Wenn wie hier ein Aufruf direkt auf der DB-Fassade erfolgt (meistens mit `select()`), wird ein Array zurückgegeben; wenn es sich um eine Methodenkette handelt, die meistens mit `get()` endet, wird eine Collection zurückgegeben.

Illuminate-Collections

Vor Laravel 5.3 lieferte die DB-Fassade ein `stdClass`-Objekt für Methoden, die nur eine Zeile – wie `first()` – zurückgeben, und ein *Array* für alle Methoden, die mehrere Zeilen – wie `all()` – zurückgeben. Ab Laravel 5.3 geben sowohl die DB-Fassade wie auch Eloquent bei allen Methoden, deren Ergebnis mehrere Zeilen umfassen kann, eine *Collection* zurück. Genauer: Die DB-Fassade liefert dabei eine Instanz von `Illuminate\Support\Collection`, während es bei Eloquent eine Instanz von `Illuminate\Database\Eloquent\Collection` ist, die im Vergleich zu `Illuminate\Support\Collection` um einige Eloquent-spezifische Methoden erweitert wurde.

Eine *Collection* (dt. *Sammlung*) könnte man als ein PHP-Array mit Superkräften bezeichnen, auf der Methoden wie `map()`, `filter()`, `reduce()`, `each()` und viele mehr ausgeführt werden können. Mehr über Collections erfahren Sie in Kapitel 17.

Parameterbindungen und benannte Bindungen

Die Datenbankarchitektur von Laravel ermöglicht die Verwendung von PDO-Parameterbindungen, um Ihre Anfragen vor möglichen SQL-Angriffen zu schützen.

Man bindet einen Parameter an eine Anweisung ganz einfach, indem man den Wert durch ein ? ersetzt und separat als zweiten Parameter hinzufügt:

```
$usersOfType = DB::select(
    'select * from users where type = ?',
    [$type]
);
```

Sie können diese Parameter aus Gründen der Übersichtlichkeit auch benennen:

```
$usersOfType = DB::select(
    'select * from users where type = :type',
    ['type' => $userType]
);
```

Direkte INSERTs

Ab hier ähneln sich die direkten Befehle alle. Direkte INSERTs sehen so aus:

```
DB::insert(
    'insert into contacts (name, email) values (?, ?)',
    ['sally', 'sally@me.com']
);
```

Direkte UPDATEs

UPDATEs schreibt man in dieser Form:

```
$countUpdated = DB::update(
    'update contacts set status = ? where id = ?',
    ['donor', $id]
);
```

Direkte DELETEs

Und DELETEs formuliert man so:

```
$countDeleted = DB::delete(
    'delete from contacts where archived = ?',
    [true]
);
```

Verkettung mit dem Query Builder

Bisher haben wir den Query Builder an sich noch gar nicht verwendet. Wir haben bloß einfache Methodenaufrufe der `DB`-Fassade benutzt. Lassen Sie uns jetzt tatsächlich einige Abfragen erstellen.

Der Query Builder ermöglicht es, Methoden miteinander zu verketten, um – es ist offensichtlich – eine Abfrage aus mehreren Elementen zusammenzusetzen. Am Ende einer Kette verwendet man meistens die `get()`-Methode, um die erstellte Abfrage dann abschließend auch tatsächlich auszuführen.

Werfen wir einen Blick auf ein kurzes Beispiel:

```
$usersOfType = DB::table('users')
    ->where('type', $type)
    ->get();
```

Hier schränken wir eine Abfrage der Tabelle `users` auf Zeilen ein, die als Eigenschaft `type` den Wert haben, der in der Variablen `$type` steht. Dann wird die Abfrage mit `get()` ausgeführt, und wir weisen unser Ergebnis der Variablen `$usersOfType` zu.

Lassen Sie uns einen Blick darauf werfen, welche Methoden man im Query Builder verketten kann. Es gibt einschränkende, modifizierende und bedingte sowie Beendigungs- und Rückgabemethoden.

Einschränkende Methoden

Diese Methoden schränken die Abfrage ein, um eine kleinere Teilmenge an Daten zu erhalten:

`select()` *und* `addSelect()`
: Mit `select()` kann man festlegen, welche Spalten zurückgegeben werden sollen; mit `addSelect()` kann man einer vorhandenen Query-Builder-Instanz weitere Spalten hinzufügen:

```
$emails = DB::table('contacts')
    ->select('email', 'email2 as second_email')
    ->get();
// oder
$emails = DB::table('contacts')
    ->select('email')
    ->addSelect('email2 as second_email')
    ->get();
```

Beide Methoden erlauben seit Version 6 bei Bedarf auch Subqueries. Mehr dazu finden Sie in der Dokumentation (*https://bit.ly/2RBgkcP*) und in einem Beitrag (*https://laravel-news.com/eloquent-subquery-enhancements*), den der Autor dieses Features verfasst hat.

`where()`
: Legt eine `WHERE`-Bedingung fest, die erfüllt werden muss: Standardmäßig erwartet die `where()`-Methode drei Parameter – den Spaltennamen, einen Vergleichsoperator und einen Vergleichswert:

```
$newContacts = DB::table('contact')
    ->where('created_at', '>', now()->subDay())
    ->get();
```

Wenn Sie mit `=` auf Gleichheit prüfen wollen, was am häufigsten vorkommt, können Sie den zweiten Operator auch weglassen:

```
$vipContacts = DB::table('contacts')->where('vip', true)->get();
```

Wenn Sie `where()`-Anweisungen kombinieren möchten, können Sie sie entweder verketten oder ein Array von Arrays übergeben:

```
$newVips = DB::table('contacts')
    ->where('vip', true)
    ->where('created_at', '>', now()->subDay());
// oder
$newVips = DB::table('contacts')->where([
    ['vip', true],
    ['created_at', '>', now()->subDay()],
]);
```

`orWhere()`

Verknüpft eine zusätzliche `WHERE`-Bedingung per `OR` mit einer bereits vorliegenden `WHERE`-Bedingung:

```
$priorityContacts = DB::table('contacts')
    ->where('vip', true)
    ->orWhere('created_at', '>', now()->subDay())
    ->get();
```

Um komplexere, mit `OR` verknüpfte `WHERE`-Anweisungen mit mehreren Bedingungen zu erstellen, können Sie eine Closure benutzen:

```
$contacts = DB::table('contacts')
    ->where('vip', true)
    ->orWhere(function ($query) {
        $query->where('created_at', '>', now()->subDay())
            ->where('trial', false);
    })
    ->get();
```

Potenzielle Fehlerquellen bei der Verwendung mehrerer where()- und orWhere()-Aufrufe

Wenn Sie `orWhere()`-Aufrufe mit mehreren `where()`-Aufrufen kombinieren, sollten Sie sicherstellen, dass die Abfrage auch wirklich das gewünschte Ergebnis liefert. Das liegt nicht an irgendeinem Fehler in Laravel, sondern einfach daran, wie in einer Abfrage die Reihenfolge der Bedingungen ausgewertet wird – zum besseren Verständnis zeige ich Ihnen auch den SQL-Befehl, zu dem Laravel die Abfrage umformuliert:

```
$canEdit = DB::table('users')
    ->where('admin', true)
    ->orWhere('plan', 'premium')
    ->where('is_plan_owner', true)
    ->get();
```

```
SELECT * FROM users
    WHERE admin = 1
    OR plan = 'premium'
    AND is_plan_owner = 1;
```

Wenn Sie einen SQL-Befehl schreiben möchten, der als Bedingung so etwas wie »WENN a ODER (b UND c)« – was im vorherigen Beispiel eindeutig die Absicht war – enthalten soll, müssen Sie an `orWhere()` eine Closure übergeben, um die erforderliche Klammerung nachzubilden:

```
$canEdit = DB::table('users')
    ->where('admin', true)
```

```
    ->orWhere(function ($query) {
        $query->where('plan', 'premium')
            ->where('is_plan_owner', true);
    })
    ->get();
```

```
SELECT * FROM users
    WHERE admin = 1
    OR (plan = 'premium' AND is_plan_owner = 1);
```

`whereBetween(`*`colName`*`, [`*`low, high`*`])`

Mit dieser Methode lässt sich eine Bedingung so beschränken, dass nur Zeilen zurückgegeben werden, bei denen der Wert einer Spalte zwischen zwei vorgegebenen Randwerten liegt (einschließlich der Randwerte):

```
$mediumDrinks = DB::table('drinks')
    ->whereBetween('size', [6, 12])
    ->get();
```

Das funktioniert auch mit `whereNotBetween()`, nur dass hier auf Werte *außerhalb* der Randwerte eingeschränkt wird.

`whereIn(`*`colName`*`, [`*`1, 2, 3`*`])`

Damit lässt sich eine Bedingung so formulieren, dass nur Zeilen zurückgegeben werden, bei denen ein Spaltenwert mit einem der Werte in einer vorgegebenen Liste übereinstimmt:

```
$closeBy = DB::table('contacts')
    ->whereIn('state', ['FL', 'GA', 'AL'])
    ->get();
```

Das funktioniert auch mit `whereNotIn()`, nur dass hier auf Werte eingeschränkt wird, die sich *nicht* in der Liste befinden.

`whereNull(`*`colName`*`)` *und* `whereNotNull(`*`colName`*`)`

Damit kann man Zeilen auswählen, in denen eine bestimmte Spalte `NULL` oder `NOT NULL` ist.

`whereRaw()`

Ermöglicht die Übergabe einer »rohen«, nicht maskierten Zeichenkette, die als Bedingung einer `WHERE`-Anweisung benutzt wird:

```
$goofs = DB::table('contacts')->whereRaw('id = 12345')->get()
```

Vorsicht vor SQL Injections!

SQL-Abfragen, die an `whereRaw()` übergeben werden, werden nicht maskiert. Verwenden Sie diese Methode mit Bedacht – dies eröffnet unter Umständen eine hervorragende Gelegenheit für SQL-Injection-Angriffe auf Ihre Anwendung.

`whereExists()`

Damit kann man eine Abfrage formulieren, die nur Zeilen zurückgibt, die die in `whereExists()` übergebene Subquery erfüllen. Stellen Sie sich beispielsweise

vor, Sie wollen nur diejenigen Benutzer selektieren, die mindestens einen Kommentar hinterlassen haben:

```
$commenters = DB::table('users')
    ->whereExists(function ($query) {
        $query->select('id')
            ->from('comments')
            ->whereRaw('comments.user_id = users.id');
    })
    ->get();
```

`distinct()`
: Beschränkt das Ergebnis auf eindeutig voneinander unterscheidbare Datensätze. Normalerweise wird diese Methode mit `select()` kombiniert – Duplikate im Abfrageergebnis werden dann durch `distinct()` eliminiert:

```
$cities = DB::table('contacts')->select('city')->distinct()->get();
```

Modifizierende Methoden

Diese Methoden ändern die Art und Weise, wie die Ergebnisse der Query *ausgegeben* werden:

`orderBy(`*`colName, direction`*`)`
: Sortiert die Ergebnisse. Der zweite Parameter kann entweder `asc` (die Standardeinstellung, aufsteigende Reihenfolge, für »ascending«) oder `desc` (absteigende Reihenfolge, für »descending«) lauten:

```
$contacts = DB::table('contacts')
    ->orderBy('last_name', 'asc')
    ->get();
```

  Diese Methode erlaubt seit Version 6 auch die Verwendung von Subqueries. Weiter oben im Absatz zu `select()` und `addSelect()` finden Sie dazu zwei weiterführende Links.

`groupBy()` *und* `having()` *oder* `havingRaw()`
: Gruppiert Ergebnisse anhand einer Spalte. Optional können Sie mit `having()` und `havingRaw()` Ihre Ergebnisse auch nach bestimmten Eigenschaften der Gruppen filtern. Sie könnten beispielsweise nur nach Orten mit mindestens 30 Einwohnern suchen:

```
$populousCities = DB::table('contacts')
    ->groupBy('city')
    ->havingRaw('count(contact_id) > 30')
    ->get();
```

`skip()` *und* `take()`
: Diese Methoden werden meist für die Paginierung von Listenausgaben verwendet, indem man festlegt, wie viele Zeilen zurückgegeben werden sollen und ab welcher Zeile damit begonnen werden soll – damit bestimmt man in

der Praxis den Umfang einzelner Listen und die Verteilung der Ergebnisse auf einzelne Seiten:

```
// Gibt die Zeilen 31–40 zurück
$page4 = DB::table('contacts')->skip(30)->take(10)->get();
```

`latest(`*`colName`*`)` *und* `oldest(`*`colName`*`)`
: Sortiert Ergebnisse nach der genannten Spalte (oder nach `created_at`, wenn kein Spaltenname übergeben wird) in absteigender Reihenfolge bei `latest()` oder aufsteigender bei `oldest()`.

`inRandomOrder()`
: Sortiert das Ergebnis in zufälliger Reihenfolge.

Bedingte Methoden

Es gibt zwei Methoden, die ab Laravel 5.2 verfügbar sind und deren »Inhalt« (eine Closure) abhängig vom booleschen Zustand eines festgelegten Werts nur bedingt angewendet wird:

`when()`
: Wenn der erste Parameter den Wahrheitswert `true` aufweist, wird die in der Closure definierte Abfragemodifikation angewendet, bei `false` nicht. Bitte beachten Sie, dass der erste Parameter ein Boolean (im folgenden Beispiel `$ignoreDrafts`, eine Variable mit dem Wert `true` oder `false`), ein optionaler Wert (im Beispiel `$status` als Benutzereingabe, mit einem Defaultwert von `null`) oder eine Closure sein kann – wichtig ist, dass der Parameter letztlich zu wahr oder falsch ausgewertet werden kann. Zum Beispiel:

```
$status = request('status'); // Mit einem Defaultwert von null, falls nicht
                             // gesetzt

$posts = DB::table('posts')
    ->when($status, function ($query) use ($status) {
        return $query->where('status', $status);
    })
    ->get();

// oder
$posts = DB::table('posts')
    ->when($ignoreDrafts, function ($query) {
        return $query->where('draft', false);
    })
    ->get();
```

Sie können auch einen dritten Parameter übergeben, und zwar eine weitere Closure, die nur ausgeführt wird, wenn der erste Parameter `false` ergibt.

`unless()`
: Das genaue Gegenteil von `when()`. Wenn der erste Parameter `false` ergibt, wird die Closure ausgeführt.

Ende-/Rückgabemethoden

Diese Methoden beenden die Methodenverkettung und führen die SQL-Abfrage aus. Fehlt am Ende der Abfragekette eine solche Methode, bekommen Sie nur eine Instanz des Query Builders zurückgeliefert; erst mit einer dieser Methoden erhalten Sie tatsächlich auch ein Abfrageergebnis:

`get()`
: Ruft alle Ergebnisse der erstellten Abfrage ab:

```
$contacts = DB::table('contacts')->get();
$vipContacts = DB::table('contacts')->where('vip', true)->get();
```

`first()` *und* `firstOrFail()`
: Damit erhält man die erste Zeile des Ergebnisses – quasi ein `get()`, dem ein `LIMIT 1` hinzugefügt wurde:

```
$newestContact = DB::table('contacts')
    ->orderBy('created_at', 'desc')
    ->first();
```

  `first()` schlägt stillschweigend fehl, wenn es kein Ergebnis gibt, während `firstOrFail()` in diesem Fall eine Exception auslöst.

  Wenn Sie diesen beiden Methoden ein Array mit Spaltennamen übergeben, liefern sie Daten nur für diese Spalten zurück.

`find(`*`id`*`)` *und* `findOrFail(`*`id`*`)`
: Wie `first()`, aber Sie können einen ID-*Wert* übergeben (keinen Spaltennamen!), der dem zu suchenden Primärschlüssel entspricht. `find()` schlägt stillschweigend fehl, wenn keine Zeile mit dieser ID existiert, während `findOrFail()` eine Exception auslöst:

```
$contactFive = DB::table('contacts')->find(5);
```

`value()`
: Pickt sich nur den Wert eines einzelnen Felds aus der ersten Zeile des Ergebnisses heraus. Wie `first()`, aber bezogen auf eine einzige Spalte:

```
$newestContactEmail = DB::table('contacts')
    ->orderBy('created_at', 'desc')
    ->value('email');
```

`count()`
: Gibt an, wie viele Zeilen ein Ergebnis enthält:

```
$countVips = DB::table('contacts')
    ->where('vip', true)
    ->count();
```

`min()` *und* `max()`
: Liefert den minimalen oder maximalen Wert einer bestimmten Spalte:

```
$highestCost = DB::table('orders')->max('amount');
```

`sum()` *und* `avg()`

Liefert die Summe oder den Durchschnitt aller Werte einer bestimmten Spalte:

```
$averageCost = DB::table('orders')
    ->where('status', 'completed')
    ->avg('amount');
```

Mit DB::raw direkte SQL-Abfragen innerhalb von Query-Builder-Methoden formulieren

Sie haben bereits einige benutzerdefinierte Methoden für direkte SQL-Abfragen kennengelernt – beispielsweise gibt es zu `select()` ein `selectRaw()`-Gegenstück, mit dem eine Zeichenkette an den Query Builder übergeben werden kann, die hinter der `WHERE`-Anweisung platziert wird.

Sie können tatsächlich an fast jede Methode im Query Builder das Ergebnis eines `DB::raw()`-Aufrufs übergeben, um das gleiche Resultat zu erzielen:

```
$contacts = DB::table('contacts')
    ->select(DB::raw('*, (score * 100) AS integer_score'))
    ->get();
```

Joins

Joins sind naturgemäß nicht ganz einfach zu definieren, aber der Query Builder hilft dabei, diesen Prozess zu vereinfachen. Betrachten wir ein Beispiel:

```
$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->select('users.*', 'contacts.name', 'contacts.status')
    ->get();
```

Die Methode `join()` erzeugt einen inneren Join. Sie können auch mehrere Joins verketten oder `leftJoin()` verwenden, um einen äußeren Join zu erhalten.

Komplexere Joins können Sie erstellen, indem Sie der `join()`-Methode eine Closure übergeben:

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

Unions

Mit `union()` oder `unionAll()` können Sie die Ergebnisse zweier Abfragen zu einer gemeinsamen Ergebnismenge zusammenfügen:

```
$first = DB::table('contacts')
    ->whereNull('first_name');
```

```
$contacts = DB::table('contacts')
    ->whereNull('last_name')
    ->union($first)
    ->get();
```

Inserts

Die `insert()`-Methode ist simpel: Übergeben Sie ein Array, um eine einzelne Zeile, oder ein Array von Arrays, um mehrere Zeilen einzufügen; und verwenden Sie `insertGetId()` anstelle von `insert()`, wenn Sie die automatisch erhöhte Primärschlüssel-ID zurückerhalten möchten:

```
$id = DB::table('contacts')->insertGetId([
    'name' => 'Abe Thomas',
    'email' => 'athomas1987@gmail.com',
]);

DB::table('contacts')->insert([
    ['name' => 'Tamika Johnson', 'email' => 'tamikaj@gmail.com'],
    ['name' => 'Jim Patterson', 'email' => 'james.patterson@hotmail.com'],
]);
```

Updates

Updates sind genauso einfach. Erstellen Sie Ihre Abfrage, verwenden Sie abschließend anstelle von `get()` oder `first()` einfach `update()` und übergeben Sie dabei ein Array von Parametern:

```
DB::table('contacts')
    ->where('points', '>', 100)
    ->update(['status' => 'vip']);
```

Sie können Spaltenwerte auch mit den Methoden `increment()` und `decrement()` schnell erhöhen (inkrementieren) und verringern (dekrementieren). Der erste Parameter ist der Spaltenname und der zweite (optionale) der Wert, um den erhöht bzw. verringert werden soll:

```
DB::table('contacts')->increment('tokens', 5);
DB::table('contacts')->decrement('tokens');
```

Deletes

Deletes sind noch einfacher. Erstellen Sie Ihre Abfrage und beenden Sie sie mit `delete()`:

```
DB::table('users')
    ->where('last_login', '<', now()->subYear())
    ->delete();
```

Sie können die Tabelle mit `truncate()` auch leeren, also alle Zeilen löschen. Dabei wird auch der aktuelle Wert der autoinkrementellen ID zurückgesetzt:

```
DB::table('contacts')->truncate();
```

JSON-Operationen

Wenn Sie JSON-Spalten einsetzen, können Sie Zeilen aktualisieren oder auswählen, indem Sie die Pfeil-Syntax verwenden, um Hierarchie-Ebenen zu durchlaufen:

```
// Alle Datensätze auswählen, bei denen die Eigenschaft "isAdmin"
// der JSON-Spalte "options" auf TRUE gesetzt ist
DB::table('users')->where('options->isAdmin', true)->get();

// Alle Datensätze aktualisieren und die Eigenschaft "verified"
// der JSON-Spalte "options" auf TRUE setzen
DB::table('users')->update(['options->isVerified', true]);
```

Hierbei handelt es sich um ein Feature, das in Version 5.3 eingeführt wurde.

Transaktionen

Falls Sie mit Datenbanktransaktionen nicht vertraut sind: Mit ihnen fasst man eine Reihe von Datenbankbefehlen zusammen, die gemeinsam ausgeführt und ggf. auch gemeinsam rückgängig gemacht werden sollen. Transaktionen werden häufig verwendet, um sicherzustellen, dass entweder *alle* oder *keine*, aber nicht nur *einige* einer Reihe von zusammenhängenden Befehlen bzw. Abfragen ausgeführt werden. Schlägt eine fehl, wird die gesamte Reihe von Abfragen rückabgewickelt.

Dementsprechend werden auch bei der Transaktionsfunktion des Query Builders alle Abfragen einer Transaktion zurückgesetzt, wenn zu irgendeinem Zeitpunkt während der Ausführung Exceptions ausgelöst werden. Nur wenn die in der Closure angegebenen Aufgaben erfolgreich abgeschlossen werden können, wird der gesamte Anweisungblock abschließend committet und nicht rückabgewickelt.

Werfen wir einen Blick auf die Beispieltransaktion in Beispiel 5-16.

Beispiel 5-16: Eine einfache Datenbanktransaktion

```
DB::transaction(function () use ($userId, $numVotes) {
    // Möglicherweise fehlschlagende Datenbankabfrage
    DB::table('users')
        ->where('id', $userId)
        ->update(['votes' => $numVotes]);

    // Weitere Abfrage, die nicht ausgeführt werden soll, falls die vorherige Abfrage
    // fehlschlägt
    DB::table('votes')
        ->where('user_id', $userId)
        ->delete();
});
```

In diesem Beispiel gehen wir davon aus, dass wir in einem früheren Prozess die Anzahl der Stimmen aus einer Tabelle `votes` für einen bestimmten Benutzer bereits zusammengefasst haben. Diese Zahl wollen wir in der Tabelle `users` zwischenspeichern und die Stimmen dann aus der Tabelle `votes` löschen. Andererseits wollen

wir die Stimmen auf keinen Fall löschen, bevor das Update in der Tabelle `users` erfolgreich war. Und schließlich darf die aktualisierte Stimmanzahl nicht in der Tabelle `users` stehen bleiben, falls die Löschung der Tabelle `votes` fehlschlägt.

Geht mit einer der beiden Abfragen etwas schief, wird die andere ebenfalls nicht durchgeführt: Das macht den »Zauber« von Datenbanktransaktionen aus.

Bitte beachten Sie, dass Sie Transaktionen auch manuell beginnen und beenden können – das gilt sowohl für Abfragen mit dem Query Builder wie für Eloquent-Abfragen. Sie beginnen eine Transaktion mit `DB::beginTransaction()`, beenden sie mit `DB::commit()` und brechen sie mit `DB::rollBack()` ab:

```
DB::beginTransaction();

// Hier gewünschte Datenbankaktionen durchführen

if ($badThingsHappened) {
    DB::rollBack();
}

// Hier weitere Datenbankaktionen durchführen

DB::commit();
```

Einführung in Eloquent

Nachdem wir nun den Query Builder kennengelernt haben, möchte ich Ihnen Eloquent vorstellen, Laravels Vorzeige-Datenbankwerkzeug, das auf dem Query Builder basiert.

Eloquent ist ein ActiveRecord-ORM – eine Abstraktionsschicht, die als Schnittstelle zu den unterschiedlichsten Datenbanktypen fungiert. Als objektrelationaler Mapper ist Eloquent dafür zuständig, zwischen den Datenobjekten innerhalb der Anwendung und den Datensätzen in Datenbanken zu vermitteln bzw. diese gegenseitig aufeinander »abzubilden«. »ActiveRecord« bedeutet, dass eine einzelne Eloquent-Klasse nicht nur dafür verantwortlich ist, mit der Tabelle als Ganzes zu interagieren (mit der Anweisung `User::all()` erhält man beispielsweise alle Benutzer), sondern auch dafür, einzelne Tabellenzeilen darzustellen (indem ein Objekt der Klasse erzeugt wird: z. B. mit `$sharon = new User`). Zusätzlich ist jede Instanz in der Lage, ihre eigene Persistenz zu verwalten: Sie versteht z. B. Anweisungen wie `$sharon->save()` oder `$sharon->delete()`.

Eloquent legt den Schwerpunkt auf Einfachheit, und wie der Rest des Frameworks stützt es sich auf die Leitregel »Konvention vor Konfiguration«, damit Sie leistungsstarke Modelle mit minimalem Code erstellen können.

Beispielsweise können Sie alle Operationen aus Beispiel 5-18 mit dem in Beispiel 5-17 definierten Modell durchführen.

Beispiel 5-17: Das denkbar einfachste Eloquent-Modell

```
<?php

use Illuminate\Database\Eloquent\Model;

class Contact extends Model {}
```

Beispiel 5-18: Operationen, die mit dem Modell aus Beispiel 5-17 ausgeführt werden können

```
// Methoden, die in einem Controller stehen würden:
public function save(Request $request)
{
    // Anhand von Benutzereingaben einen neuen Kontakt erzeugen und speichern
    $contact = new Contact();
    $contact->first_name = $request->input('first_name');
    $contact->last_name = $request->input('last_name');
    $contact->email = $request->input('email');
    $contact->save();

    return redirect('contacts');
}

public function show($contactId)
{
    // Rückgabe einer JSON-Repräsentation eines Kontakts, basierend auf einem
    // URL-Segment
    // Wenn der Kontakt nicht existiert, wird eine Ausnahme geworfen
    return Contact::findOrFail($contactId);
}

public function vips()
{
    // Ein unnötig komplexes Beispiel, aber trotzdem umsetzbar mit einer grundlegenden
    // Eloquent-Klasse; fügt einem VIP-Eintrag die Eigenschaft "formalName" hinzu
    return Contact::where('vip', true)->get()->map(function ($contact) {
        $contact->formalName = "Der erhabene {$contact->first_name} des Geschlechts der
         {$contact->last_name}s";

        return $contact;
    });
}
```

Wie funktioniert es? Durch Konvention. Eloquent nimmt den Klassennamen des Modells (`Contact`) und erzeugt daraus – indem der englische Plural gebildet wird – den Tabellenamen (`contacts`), der natürlich in der Datenbank vorhanden sein muss, und schon haben Sie ein voll funktionsfähiges Eloquent-Modell.

Schauen wir uns jetzt an, wie man mit Eloquent-Modellen arbeitet.

Erstellen und Definieren von Eloquent-Modellen

Lassen Sie uns zuerst ein Modell erstellen. Dafür existiert ein Artisan-Befehl:

```
php artisan make:model Contact
```

Damit erzeugen wir eine Datei *app/Contact.php*, die die Modell-Klasse enthält:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    //
}
```

Eine Migration zusammen mit dem Modell anlegen

Wenn Sie beim Erstellen Ihres Modells auch automatisch eine Migration anlegen möchten, übergeben Sie das Flag `-m` oder `--migration`:

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

Tabellenname

Den Tabellennamen legt Laravel fest, indem es den Klassennamen (üblicherweise in »PascalCase« notiert) in »snake_case« umwandelt und ihn pluralisiert, sodass beispielsweise eine Migration zu einem Modell `SecondaryContact` eine Tabelle namens `secondary_contacts` erstellen würde. Wenn Sie den Tabellennamen anpassen möchten, können Sie im Modell explizit die Eigenschaft `$table` setzen:

```
protected $table = 'contacts_secondary';
```

Primärschlüssel

Laravel geht standardmäßig davon aus, dass jede Tabelle einen ganzzahligen Primärschlüssel besitzt, der automatisch inkrementiert und `id` benannt wird.

Wenn Sie den Namen des Primärschlüssels ändern möchten, ändern Sie die Modell-Eigenschaft `$primaryKey`:

```
protected $primaryKey = 'contact_id';
```

Falls der Primärschlüssel nicht inkrementell sein soll:

```
public $incrementing = false;
```

In Version 6 ist die Performance für Integer-Indizes verbessert worden. Falls der Primärschlüssel eines Modells vom Zeichenketten-Typ sein sollte, deklarieren Sie deshalb bitte im entsprechenden Modell den `$keyType`:

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

Zeitstempel

Eloquent erwartet, dass jede Tabelle die Zeitstempelspalten `created_at` und `updated_at` enthält. Wenn es diese Spalten in Ihrer Tabelle nicht gibt, ändern Sie bitte die Eigenschaft `$timestamps`:

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

Sie können das Format der Zeitstempel anpassen, indem Sie die Klasseneigenschaft `$dateFormat` auf eine benutzerdefinierte Zeichenkette setzen. Diese Zeichenkette wird mit der bekannten Syntax der PHP-Funktion `date()` analysiert, sodass das folgende Beispiel das Datum in Sekunden seit Beginn der Unix-Epoche speichern würde:

```
protected $dateFormat = 'U';
```

Abrufen von Daten mit Eloquent

Um mit Eloquent Einträge aus einer Datenbank zu holen, verwendet man in der Regel statische Methoden des Modells.

Beginnen wir damit, eine ganze Tabelle einzulesen:

```
$allContacts = Contact::all();
```

Das war einfach. Lassen Sie uns das Ergebnis ein wenig filtern:

```
$vipContacts = Contact::where('vip', true)->get();
```

Eloquent erlaubt es, Einschränkungen zu verketten, und diese Einschränkungen kommen sehr vertraut daher:

```
$newestContacts = Contact::orderBy('created_at', 'desc')
    ->take(10)
    ->get();
```

Sobald man sich über den ursprünglichen Klassennamen hinausbewegt, arbeitet man mit Laravels Query Builder. Sie können noch viel mehr machen (wir kommen gleich dazu), aber zuerst einmal ist wichtig zu wissen, dass all das, was mit dem Query Builder auf der `DB`-Fassade möglich ist, auch mit Eloquent-Modellen funktioniert.

Eine Zeile erhalten

Wie wir bereits zuvor gesehen haben, können Sie `first()` verwenden, um nur den ersten Datensatz einer Abfrage, oder `find()`, um nur den Datensatz mit der angegebenen ID zu erhalten. Benutzt man stattdessen die »OrFail«-Varianten dieser Methoden, wird eine Ausnahme ausgelöst, wenn es keine übereinstimmenden Ergebnisse gibt. `findOrFail()` wird häufig genutzt, um eine Entität anhand eines übergebenen URL-Segments zu suchen oder eine Exception auszulösen, wenn kein passender Datenbankeintrag existiert, wie Sie in Beispiel 5-19 sehen können.

Beispiel 5-19: Verwendung einer OrFail()-Methode

```
// ContactController
public function show($contactId)
{
```

```
    return view('contacts.show')
        ->with('contact', Contact::findOrFail($contactId));
}
```

Methoden wie `first()`, `firstOrFail()`, `find()` oder `findOrFail()`, die (maximal) einen einzelnen Datensatz zurückgeben, liefern eine Instanz der entsprechenden Klasse. `Contact::first()` gibt also eine Instanz der Klasse `Contact` mit den Daten aus der ersten Zeile der Tabelle zurück.

Exceptions

Wie Sie in Beispiel 5-19 sehen können, müssen wir uns in den Controllern nicht selbst um die Behandlung von Ausnahmen kümmern, die ein Eloquent-Modell auslöst, falls kein Ergebnis gefunden wird (`Illuminate\Database\Eloquent\ModelNotFoundException`). Laravels Routing-System fängt die geworfene Exception und wird automatisch eine 404-Meldung (»Not found«) im Browser anzeigen.

Sie können diese Ausnahme bei Bedarf natürlich auch selbst abfangen und behandeln.

Mehrere Zeilen erhalten

`get()` arbeitet bei Eloquent genauso wie bei normalen Aufrufen des Query Builders – Sie erstellen eine Abfrage und rufen zum Schluss `get()` auf, um die Ergebnisse zu erhalten:

```
$vipContacts = Contact::where('vip', true)->get();
```

Es gibt allerdings eine reine Eloquent-Methode, `all()`, die Ihnen begegnen kann, falls jemand eine ungefilterte Liste aller Daten in einer Tabelle erhalten möchte:

```
$contacts = Contact::all();
```

get() anstelle von all() nutzen

Überall dort, wo man `all()` verwenden kann, lässt sich auch `get()` einsetzen. `Contact::get()` liefert die gleiche Antwort wie `Contact::all()`. Sobald Sie eine Abfrage aber weiter verändern wollen, indem Sie z. B. einen `where()`-Filter hinzufügen, wird das bei `all()` nicht funktionieren, bei `get()` dagegen schon.

So verbreitet `all()` auch eingesetzt wird, ich würde empfehlen, grundsätzlich `get()` zu verwenden und die Existenz von `all()` ganz aus dem Gedächtnis zu streichen.

Ein weiterer Unterschied zwischen diesen beiden Methoden bestand bis zu Version 5.3 darin, dass `get()` ein Array von Modellen zurückgab. Seitdem sind es allerdings auch bei `get()` Collections.

Ergebnisse mit chunk() unterteilen

Wenn man große Mengen von Datensätzen (Tausende oder mehr) gleichzeitig verarbeiten muss, tauchen möglicherweise Speicherprobleme oder Blockaden auf. Mit

Laravel kann man Abfrageergebnisse in kleinere Segmente (Chunks) aufteilen und in Chargen verarbeiten, sodass die Speicherauslastung relativ moderat bleibt. Beispiel 5-20 veranschaulicht die Verwendung von `chunk()`, um ein Abfrageergebnis in Abschnitte von jeweils 100 Datensätzen aufzuteilen.

Beispiel 5-20: Chunking einer Eloquent-Abfrage, um den Speicherbedarf zu begrenzen

```
Contact::chunk(100, function ($contacts) {
    foreach ($contacts as $contact) {
        // Hier wird mit $contact weitergearbeitet
    }
});
```

Aggregate

Die Aggregatfunktionen, die im Query Builder verfügbar sind, lassen sich auch bei Eloquent-Abfragen verwenden. Zum Beispiel:

```
$countVips = Contact::where('vip', true)->count();
$sumVotes = Contact::sum('votes');
$averageSkill = User::avg('skill_level');
```

Inserts und Updates mit Eloquent

Beim Einfügen und Aktualisieren von Werten weicht Eloquent allerdings von der normalen Syntax des Query Builders ab.

Inserts

Es gibt zwei grundlegende Möglichkeiten, einen neuen Datensatz mit Eloquent-Modellen hinzuzufügen.

Erstens können Sie eine neue Instanz der Eloquent-Klasse erstellen, deren Eigenschaften manuell festlegen und diese Instanz mit `save()` speichern wie in Beispiel 5-21.

Beispiel 5-21: Einfügen eines Datensatzes durch Erstellen einer neuen Instanz

```
$contact = new Contact;
$contact->name = 'Ken Hirata';
$contact->email = 'ken@hirata.com';
$contact->save();

// oder

$contact = new Contact([
    'name' => 'Ken Hirata',
    'email' => 'ken@hirata.com',
]);
$contact->save();

// oder
```

```
$contact = Contact::make([
    'name' => 'Ken Hirata',
    'email' => 'ken@hirata.com',
]);
$contact->save();
```

Bis Sie `save()` tatsächlich aufrufen, repräsentiert diese Instanz von `Contact` zwar vollständig den neuen Kontakt – aber er ist bis dahin noch nicht in der Datenbank gespeichert worden. Das bedeutet, die Instanz hat noch keine `id`, bleibt nicht erhalten, wenn die Anwendung beendet wird, und die Zeitstempel `created_at` und `updated_at` sind natürlich auch noch nicht gesetzt.

Sie können alternativ auch ein Array an `Model::create()` übergeben, wie in Beispiel 5-22 gezeigt. Im Gegensatz zu `make()` speichert `create()` die Instanz sofort in der Datenbank, es ist kein separater Aufruf von `save()` nötig.

Beispiel 5-22: Einfügen eines Datensatzes durch Übergabe eines Arrays an create()

```
$contact = Contact::create([
    'name' => 'Keahi Hale',
    'email' => 'halek481@yahoo.com',
]);
```

Bitte beachten Sie auch, dass immer dann, wenn Sie ein Array an eine Methode übergeben, die eine Instanz des Modells erzeugt und/oder den Datensatz speichert oder ändert (also `new Model()`, `Model::make()`, `Model::create()` oder `Model::update()`), alle betroffenen Eigenschaften für die »Massenzuweisung« (»mass assignment«) freigegeben sein müssen. Das ist bei der ersten Variante in Beispiel 5-21 nicht nötig, da dort die Eigenschaften einzeln zugewiesen werden und nicht innerhalb eines Arrays.

Updates

Die Aktualisierung von Datensätzen ähnelt dem Einfügen. Sie können eine bestimmte Instanz abrufen, deren Eigenschaften ändern und die Instanz dann erneut speichern, oder Sie erledigen alles in einem einzelnen Aufruf und übergeben ein Array mit aktualisierten Eigenschaften. Beispiel 5-23 veranschaulicht den ersten Ansatz.

Beispiel 5-23: Instanz aktualisieren und Datensatz speichern

```
$contact = Contact::find(1);
$contact->email = 'natalie@parkfamily.com';
$contact->save();
```

Da dieser Datensatz bereits existiert, besitzt er bereits einen `created_at`-Zeitstempel und eine `id`, die natürlich beide nicht verändert werden, während das Feld `updated_at` auf das aktuelle Datum und die aktuelle Uhrzeit gesetzt wird. Beispiel 5-24 veranschaulicht den zweiten Ansatz.

Beispiel 5-24: Aktualisieren von Datensätze durch Übergabe eines Arrays an die update()-Methode

```
Contact::where('created_at', '<', now()->subYear())
    ->update(['longevity' => 'ancient']);

// oder

$contact = Contact::find(1);
$contact->update(['longevity' => 'ancient']);
```

Die `update()`-Methode erwartet ein Array, in dem die Schlüssel die Spaltennamen enthalten und die Werte die neuen Eigenschaften.

Massenzuweisung

Wir haben uns bereits einige Beispiele angesehen, in denen Arrays mit Werten vorkommen, die an die Modell-Methoden übergeben werden. Solange Sie nicht definiert haben, welche Felder eines Modells durch sogenannte Massenzuweisungen füllbar bzw. *fillable* sind, wird das allerdings noch nicht funktionieren.

Das dient als Schutz vor (möglicherweise bösartigen) Benutzereingaben, durch die neue Werte für Felder gesetzt werden könnten, die gar nicht geändert werden sollen. Betrachten Sie das häufig vorkommende Szenario in Beispiel 5-25.

Beispiel 5-25: Aktualisieren eines Eloquent-Modells anhand der gesamten übergebenen Benutzereingabe

```
// ContactController
public function update(Contact $contact, Request $request)
{
    $contact->update($request->all());
}
```

Falls Sie mit dem `Request`-Objekt noch nicht vertraut sind: In Beispiel 5-25 wird die gesamte in diesem Objekt transportierte Benutzereingabe genommen und an die `update()`-Methode übergeben. Das Ergebnis der `all()`-Methode beinhaltet auch Dinge wie URL-Parameter und Formulareingaben, sodass ein bösartiger Benutzer manipulative Angaben hinzufügen könnte, z.B. Daten wie `id` und `owner_id`, die höchstwahrscheinlich nicht verändert werden sollen.

Glücklicherweise wird das verhindert, solange die zu aktualisierenden Eigenschaften des Modells nicht als massenzuweisungstauglich gekennzeichnet wurden. Dazu kann man entweder die erlaubten Felder in einer Whitelist-Variablen `$fillable` aufführen oder die verbotenen und damit geschützten Felder einer Blacklist-Variablen `$guarded` hinzufügen, um festzulegen, welche Felder per Massenzuweisung (*mass assignment*) bearbeitet werden können und welche nicht. Eine Massenzuweisung liegt immer dann vor, wenn Sie ein Array von Werten an `create()` oder `update()` übergeben. Bitte beachten Sie, dass Eigenschaften, die nicht als *fillable*

gekennzeichnet sind, dennoch weiterhin durch direkte Zuweisungen geändert werden können (z. B. mit Anweisungen wie `$contact->password='abc'`). Beispiel 5-26 zeigt beide Ansätze.

Beispiel 5-26: Massenzuweisungen von Eigenschaften erlauben bzw. verbieten

```
class Contact
{
    protected $fillable = ['name', 'email'];

    // oder

    protected $guarded = ['id', 'created_at', 'updated_at', 'owner_id'];
}
```

Massenzuweisung mit Request::only()

In Beispiel 5-25 wäre der Massenzuweisungsschutz von Eloquent wichtig gewesen, weil wir darin die `all()`-Methode auf das `Request`-Objekt anwenden, um die *Gesamtheit* aller Benutzereingaben zu speichern.

Der Massenzuordnungsschutz von Eloquent ist hier eine hervorragende Hilfe, aber es gibt daneben einen hilfreichen Trick, um die Eingaben eines Benutzers zu filtern.

Die Klasse `Request` besitzt eine Methode namens `only()`, mit der man die Schlüssel beschränken kann, die aus der Benutzereingabe übernommen werden sollen. Damit könnten Sie z. B. formulieren:

```
Contact::create($request->only('name', 'email'));
```

firstOrCreate() und firstOrNew()

Manchmal möchte man seiner Anwendung den Befehl erteilen: »Gib mir eine Instanz mit genau diesen Eigenschaften oder erstelle sie, wenn sie noch nicht existiert.« Hier kommen die `firstOr*()`-Methoden ins Spiel.

Die beiden Methoden `firstOrCreate()` und `firstOrNew()` erwarten ein Array von Schlüssel/Wert-Paaren als ersten Parameter:

```
$contact = Contact::firstOrCreate(['email' => 'luis.ramos@myacme.com']);
```

Beide Methoden suchen nach dem ersten Datensatz, der den Vorgaben entspricht, und erzeugen eine neue Instanz mit den gewünschten Eigenschaften, falls es keine übereinstimmenden Datensätze gibt; `firstOrCreate()` wird diese Instanz in der Datenbank speichern und dann zurückgeben, während `firstOrNew()` sie zwar erzeugt, aber nicht speichert.

Wenn Sie ein weiteres Array von Werten als zweiten Parameter übergeben, werden diese Werte einem neu erstellten Eintrag hinzugefügt, aber *nicht* verwendet, um den Eintrag zu suchen.

Löschen mit Eloquent

Das Löschen mit Eloquent ähnelt stark dem Aktualisieren. Mit der Option, Datensätze per Soft Delete nur »weich« zu löschen, können die Daten sogar für eine spätere Überprüfung oder Wiederherstellung archiviert werden.

Normales Löschen

Der einfachste Weg, einen Datensatz zu löschen, ist der Aufruf der Methode `delete()` auf der Instanz selbst:

```
$contact = Contact::find(5);
$contact->delete();
```

Wenn Sie die ID bereits kennen, gibt es allerdings keinen Grund, eine Instanz erst nachzuschlagen; Sie können direkt eine ID oder ein Array von IDs an die `destroy()`-Methode des Modells übergeben:

```
Contact::destroy(1);
// oder
Contact::destroy([1, 5, 7]);
```

Sie können auch alle Ergebnisse einer Abfrage löschen:

```
Contact::where('updated_at', '<', now()->subYear())->delete();
```

Soft Deletes oder »weiches« Löschen

Das »weiche« Löschen markiert Einträge in der Datenbank zwar als gelöscht, entfernt sie aber nicht endgültig aus der Datenbank. Das eröffnet die Möglichkeit, die Daten nachträglich zu kontrollieren, sie bei der Anzeige historischer Informationen mit zu berücksichtigen oder Benutzern (oder Administratoren) die Gelegenheit zu geben, einige oder alle gelöschten Daten wiederherzustellen.

Würde man eine Anwendung mit Soft Deletes manuell programmieren, läge die Herausforderung darin, die nur scheinbar gelöschten Daten bei jeder einzelnen Abfrage auszuschließen. Glücklicherweise wird in Laravel jede Abfrage »unter der Haube« automatisch so angepasst, dass weich gelöschte Daten ignoriert werden – es sei denn, die Abfrage wird bewusst so formuliert, dass sie doch berücksichtigt werden sollen. Soft Deletes funktionieren nur, wenn die zu einem Modell gehörige Tabelle eine Spalte `deleted_at` besitzt.

Wann sollte man Soft Deletes verwenden?

Nur weil ein Feature existiert, bedeutet das nicht, dass Sie es auch verwenden sollten. Viele Laravel-Programmierer setzen Soft Deletes standardmäßig in jedem Projekt ein, bloß weil die Funktion vorhanden ist. Der Einsatz von Soft Deletes hat

jedoch auch Nachteile. Es ist ziemlich wahrscheinlich, dass Sie gelegentlich vergessen, an die Spalte `deleted_at` zu denken, wenn Sie mit Tools wie Sequel Pro oder phpMyAdmin Ihre Datenbank direkt bearbeiten oder abfragen. Und solange Sie »weich« gelöschte Datensätze nicht endgültig entfernen, werden Ihre Datenbanken schneller wachsen als beim echten Löschen alter Datensätze.

Deshalb meine Empfehlung: Benutzen Sie Soft Deletes nicht standardmäßig. Verwenden Sie sie stattdessen nur, wenn Sie sie wirklich brauchen. Und wenn Sie es tun, entfernen Sie alte, weich gelöschte Datensätze so offensiv wie möglich mit einem Tool wie Quicksand (*https://github.com/tightenco/quicksand*). Soft Deletes sind ein leistungsstarkes Feature, aber der Einsatz ist nicht immer und von vornherein sinnvoll.

Soft Deletes aktivieren. Um Soft Deletes für ein Modell/eine Tabelle zu aktivieren, müssen Sie drei Dinge erledigen: die Spalte `deleted_at` der Migration der Eigenschaft `$dates` hinzufügen sowie den Trait `SoftDeletes` ins Modell importieren. In der Schema-Fassade gibt es eine Methode, mit der man die Spalte `deleted_at` einer Tabelle hinzufügen kann, wie Sie in Beispiel 5-27 sehen können. Und Beispiel 5-28 zeigt ein Eloquent-Modell mit aktivierten Soft Deletes.

Beispiel 5-27: Eine Migration, die einer Tabelle Soft Deletes hinzufügt

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

Beispiel 5-28: Ein Eloquent-Modell mit aktivierten Soft Deletes

```
<?php

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

class Contact extends Model
{
    use SoftDeletes; // den Trait einbinden

    protected $dates = ['deleted_at']; // die Spalte als Datum registrieren
}
```

Sobald Sie diese Änderungen vorgenommen haben, setzt jeder Aufruf von `delete()` und `destroy()` die Spalte `deleted_at` der betroffenen Zeilen auf das aktuelle Datum und die aktuelle Uhrzeit, anstatt diese Zeile tatsächlich zu löschen. Sobald `deleted_at` einen Wert enthält, gilt eine Zeile als (weich) gelöscht. Und alle zukünftigen Abfragen werden diese Zeilen standardmäßig vom Ergebnis ausschließen.

Weich gelöschte Datensätze abfragen. Aber wie kommen wir jetzt bei Bedarf noch an die per Soft Delete gelöschten Datensätze?

Erstens können Sie eine Abfrage so erweitern, dass auch weich gelöschte Elemente berücksichtigt werden:

```
$allHistoricContacts = Contact::withTrashed()->get();
```

Zweitens können Sie die Methode `trashed()` verwenden, um zu prüfen, ob eine bestimmte Instanz weich gelöscht wurde:

```
if ($contact->trashed()) {
    // Hier Anweisungen ...
}
```

Schließlich können Sie auch *nur* die weich gelöschten Einträge abfragen:

```
$deletedContacts = Contact::onlyTrashed()->get();
```

Wiederherstellen weich gelöschter Elemente. Wenn Sie ein weich gelöschtes Element wiederherstellen möchten, können Sie `restore()` auf einer Instanz oder einer Abfrage ausführen:

```
$contact->restore();

// oder

Contact::onlyTrashed()->where('vip', true)->restore();
```

Weich gelöschte Elemente endgültig entfernen. Sie können weich gelöschte Datensätze endgültig aus der Datenbank entfernen, indem Sie `forceDelete()` auf einer Instanz oder Abfrage ausführen:

```
$contact->forceDelete();

// oder

Contact::onlyTrashed()->forceDelete();
```

Geltungsbereiche

Wir haben jetzt bereits viele »gefilterte« Querys benutzt, bei denen wir nur einen Teil des Tabelleninhalts abgefragt haben. Allerdings haben wir die Filterung bisher jedes Mal als manuellen Prozess mithilfe des Query Builders durchgeführt.

Stattdessen kann man auch lokale oder globale Geltungsbereiche definieren. Damit setzt man vordefinierte »Scopes« (bzw. Filter) ein, die – global – bei allen Abfragen eines Modell berücksichtigt werden oder aber – lokal – bei bestimmten Methodenketten.

Lokale Geltungsbereiche

Lokale Geltungsbereiche sind am einfachsten zu verstehen. Nehmen wir dieses Beispiel:

```
$activeVips = Contact::where('vip', true)->where('trial', false)->get();
```

Wenn wir solch eine Kette von Abfragemethoden immer wieder benutzen müssen, wird es irgendwann mühsam. Zudem würde eine derartige Definition eines »aktiven VIPs« an vielen unterschiedlichen Stellen im Code vorkommen, was bei anstehenden Änderungen eindeutig nachteilig wäre. Lässt sich diese Definition irgendwie zentralisieren? Was wäre, wenn wir es einfach so schreiben könnten:

```
$activeVips = Contact::activeVips()->get();
```

Können wir! Mithilfe eines lokalen Geltungsbereichs. Der sich einfach in der Klasse `Contact` definieren lässt, wie Sie in Beispiel 5-29 sehen können.

Beispiel 5-29: Definition eines lokalen Geltungsbereichs für ein Modell

```
class Contact
{
    public function scopeActiveVips($query)
    {
        return $query->where('vip', true)->where('trial', false);
    }
```

Um einen lokalen Geltungsbereich (bzw. *Scope*) zu definieren, fügen wir der Eloquent-Klasse eine Methode hinzu, die mit »scope« beginnt und dann den gewünschten Namen des Geltungsbereichs enthält. Diese Methode erwartet ein Query-Builder-Objekt und gibt auch eines zurück, aber innerhalb der Methode können Sie die Query natürlich verändern – das ist der springende Punkt.

Sie können auch Scopes definieren, die Parameter akzeptieren, wie Beispiel 5-30 zeigt.

Beispiel 5-30: Parameterübergabe an Scopes

```
class Contact
{
    public function scopeStatus($query, $status)
    {
        return $query->where('status', $status);
    }
```

Beim Aufruf muss dann einfach der Parameter an den Geltungsbereich übergeben werden:

```
$friends = Contact::status('friend')->get();
```

Globale Geltungsbereiche

Bei den Soft Deletes haben wir ja festgestellt, dass sie nur funktionieren, weil bei *jeder Abfrage* eines Modells, bei dem Soft Deletes aktiviert sind, diese weich gelöschten Elemente automatisch ignoriert werden. Da ist eindeutig ein globaler Geltungsbereich am Werk. Und genauso können wir eigene globale Geltungsbereiche definieren, die bei allen Abfragen angewendet werden, die ein bestimmtes Modell betreffen.

Es gibt zwei Möglichkeiten, solch einen globalen Scope zu definieren: mit einer Closure oder einer eigenen Klasse. In beiden Fällen registrieren Sie den definierten Scope in der `boot()`-Methode des Modells. Beginnen wir in Beispiel 5-31 mit der Closure-Variante:

Beispiel 5-31: Hinzufügen eines globalen Scopes per Closure

```
...
class Contact extends Model
{
    protected static function boot()
    {
        parent::boot();

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

So einfach ist das! Wir haben gerade einen globalen Geltungsbereich namens `active` hinzugefügt: Dadurch wird ab sofort jede Abfrage auf diesem Modell auf Zeilen beschränkt, bei denen `active` den Wahrheitswert `true` hat.

Als Nächstes benutzen wir die alternative Variante, wie in Beispiel 5-32 gezeigt. Dazu erstellen wir eine Klasse, die das Interface `Illuminate\Database\Eloquent\Scope` implementiert. Wir müssen dementsprechend eine Methode `apply()` definieren, die Instanzen eines Query Builders und des Modells verwendet und in der wir der Query eine zusätzliche WHERE-Bedingung hinzufügen.

Beispiel 5-32: Erstellen einer globalen Scope-Klasse

```
<?php

namespace App\Scopes;

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

class ActiveScope implements Scope
{
    public function apply(Builder $builder, Model $model)
    {
```

```
        return $builder->where('active', true);
    }
}
```

Um diesen Bereich auf ein Modell anzuwenden, erweitern wir wieder die `boot()`-Methode der Elternklasse Model und rufen auf der Klasse statisch `addGlobalScope()` auf, wie in Beispiel 5-33 gezeigt.

Beispiel 5-33: Anwenden eines klassenbasierten globalen Scopes

```
<?php

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

class Contact extends Model
{
    protected static function boot()
    {
        parent::boot();

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

Fehlender Namensraum!
Sie haben vielleicht bemerkt, dass in einigen Beispielen die Klasse `Contact` vorkommt, aber ohne Nennung eines Namensraums. Das ist regelwidrig, und ich habe es nur aus Platzgründen gemacht. Normalerweise müssen auch die Modelle aus dem App-Namespace explizit mit ihrem Namensraum, beispielsweise `App\Contact`, angesprochen bzw. eingebunden werden.

Globale Geltungsbereiche ignorieren. Es gibt drei Möglichkeiten, bei einer Abfrage einen globalen Geltungsbereich zu ignorieren, und bei allen werden die Methoden `withoutGlobalScope()` bzw. `withoutGlobalScopes()` verwendet. Wenn Sie einen Closure-basierten Bereich ignorieren wollen, benutzen Sie einfach dessen Namen:

```
$allContacts = Contact::withoutGlobalScope('active')->get();
```

Mit `withoutGlobalScope()` oder `withoutGlobalScopes()` können Sie einen einzelnen klassenbasierten globalen Bereich ignorieren:

```
Contact::withoutGlobalScope(ActiveScope::class)->get();

Contact::withoutGlobalScopes([ActiveScope::class, VipScope::class])->get();
```

Und schließlich können Sie in einer Abfrage auch einfach alle globalen Bereiche ignorieren, indem Sie den Parameter weglassen:

```
Contact::withoutGlobalScopes()->get();
```

Anpassen von Feldinteraktionen durch Akzessoren, Mutatoren und Attribut-Casting

Nachdem wir nun besprochen haben, wie Sie mit Eloquent Datensätze in die Datenbank schreiben und sie auslesen können, widmen wir uns jetzt der Dekoration und Manipulation einzelner Modell-Attribute.

Akzessoren, Mutatoren und Attribut-Casting ermöglichen es Ihnen, die Art und Weise anzupassen, wie einzelne Attribute von Eloquent-Instanzen ein- oder ausgegeben werden. Ansonsten wird jedes Attribut einfach als Zeichenkette behandelt. Und Sie können Attribute nur exakt so verwenden, wie sie in der Datenbank vorhanden sind. Aber das können wir ändern.

Akzessoren

Akzessoren ermöglichen es Ihnen gewissermaßen, benutzerdefinierte Attribute für Ihre Modelle anzulegen, wenn Sie Daten aus der Modellinstanz *lesen*. Vielleicht wollen Sie die Darstellung einer bestimmten Spalte ändern oder ein Attribut erstellen, das in der Datenbanktabelle überhaupt nicht vorhanden ist.

Sie definieren einen Akzessor, indem Sie einem Modell eine Methode mit der folgenden Struktur hinzufügen: `get`*`{PascalCasedPropertyName}`*`Attribute`. Wenn Ihr Eigenschaftsname also `first_name` hieße, bekäme die Zugriffsmethode – eine andere Bezeichnung für einen Akzessor – den Namen `getFirstNameAttribute`.

Probieren wir es aus. Zuerst werden wir in Beispiel 5-34 eine bereits existierende Spalte so dekorieren, dass ein Defaulttext ausgegeben wird, falls sie keinen Wert enthält.

Beispiel 5-34: Dekorieren einer bereits vorhandenen Spalte durch eine Zugriffsmethode

```
// Modelldefinition:
class Contact extends Model
{
    public function getNameAttribute($value)
    {
        return $value ?: '(No name provided)';
    }
}

// Verwendung des Akzessors:
$name = $contact->name;
```

Aber wir können Akzessoren auch verwenden, um Attribute zu definieren, die in der Datenbank in dieser Form überhaupt nicht existieren, wie in Beispiel 5-35.

Beispiel 5-35: Definition eines neuen Attributs per Zugriffsmethode

```
// Modelldefinition:
class Contact extends Model
{
    public function getFullNameAttribute()
```

```
    {
        return $this->first_name . ' ' . $this->last_name;
    }
}

// Verwendung des Akzessors:
$fullName = $contact->full_name;
```

Mutatoren

Mutatoren funktionieren genauso wie Akzessoren, nur dass man mit ihnen festlegt, wie die Daten vor dem *Schreiben*, nicht wie sie nach dem *Lesen* verändert werden sollen. Man bezeichnet Akzessoren im Programmierer-Jargon häufig auch als Getter und Mutatoren als Setter, basierend auf den entsprechenden englischen Verben für diese Vorgänge. Genau wie bei den Akzessoren können Sie sie verwenden, um Daten verändert in bestehende Spalten zu schreiben oder um mit Attributen arbeiten zu können, denen keine Spalten in der Datenbank entsprechen.

Sie definieren einen Mutator, indem Sie einem Modell eine Methode mit der folgenden Struktur hinzufügen: `set`*`{PascalCasedPropertyName}`*`Attribute`. Wenn die Eigenschaft also `first_name` hieße, bekäme der Mutator den Namen `setFirstName Attribute`.

Probieren wir es wieder aus. Zuerst werden wir in Beispiel 5-36 den Update-Wert für eine bereits bestehende Spalte auf 0 setzen, falls er negativ sein sollte.

Beispiel 5-36: Dekorieren eines Attributwerts per Mutator

```
// Definition des Mutators
class Order extends Model
{
    public function setAmountAttribute($value)
    {
        $this->attributes['amount'] = $value > 0 ? $value : 0;
    }
}

// Verwendung des Mutators
$order->amount = '15';
```

Mutatoren legen fest, wie Daten aussehen bzw. was sie enthalten sollen, indem auf die Attribute mit `$this->attributes` und der Angabe des Eigenschaftsnamens zugegriffen wird.

Definieren wir nun einen Mutator, der nach einer nicht vorhandenen Spalte `work group_name` benannt ist, aber intern eine andere Eigenschaft verändert, wie in Beispiel 5-37 gezeigt.

Beispiel 5-37: Ein Mutator, der ein »fremdes« Attribut verändert

```
// Definition des Mutators
class Order extends Model
{
```

```
    public function setWorkgroupNameAttribute($workgroupName)
    {
        $this->attributes['email'] = "{$workgroupName}@ourcompany.com";
    }
}

// Verwendung des Mutators
$order->workgroup_name = 'jstott';
```

Es ist relativ ungewöhnlich und potenziell verwirrend, einen Mutator für eine in der Datenbank nicht existierende Spalte zu erstellen (`workgroup_name`), die letztlich eine andere (`email`) verändert – aber es ist möglich ...

Attribut-Casting

Vielleicht denken Sie jetzt daran, Zugriffsmethoden zu schreiben, um bestimmte Felder als Zahlen statt als Zeichenkette auszugeben, oder Inhalte als JSON zu codieren und decodieren, um sie in einer `TEXT`-Spalte zu speichern, oder `TINYINT`-Werte von `0` und `1` zu und aus booleschen Werten zu konvertieren?

Glücklicherweise gibt es dafür in Eloquent bereits ein Feature. Mit dem sogenannten *Attribut-Casting* (bzw. Attributumwandlung, analog zur Typumwandlung von z. B. Variablen) kann man festlegen, dass Spalten immer als bestimmter Datentyp behandelt werden sollen, sowohl beim Lesen als auch beim Schreiben. Die dabei möglichen Angaben zum Datentyp sehen Sie in Tabelle 5-1.

Tabelle 5-1: Mögliche Datentypen beim Attribut-Casting

Datentyp	Beschreibung
`int|integer`	Wird als PHP (`int`) ausgegeben
`real|float|double`	Wird als PHP (`float`) ausgegeben
`string`	Wird als PHP (`string`) ausgegeben
`bool|boolean`	Wird als PHP (`bool`) ausgegeben
`object`	Wird zu/aus JSON geparst, als `stdClass`-Objekt
`array`	Wird zu/aus JSON geparst, als Array
`collection`	Wird zu/aus JSON geparst, als Collection
`date|datetime`	Wird aus einer `DATETIME`-Spalte zu Carbon und zurück geparst
`timestamp`	Wird aus einer `TIMESTAMP`-Spalte zu Carbon und zurück geparst

Beispiel 5-38 zeigt, wie Sie Attribut-Casting in einem Modell definieren.

Beispiel 5-38: Festlegen von Attribut-Casting für ein Modell

```
class Contact
{
    protected $casts = [
        'vip' => 'boolean',
        'children_names' => 'array',
```

```
        'birthday' => 'date',
    ];
}
```

Datumsmutatoren

Sie können bestimmte Spalten als `timestamp`-Spalten definieren, indem Sie sie zum `$dates`-Array hinzufügen wie in Beispiel 5-39.

Beispiel 5-39: Definieren von Spalten, die als Zeitstempel behandelt werden sollen

```
class Contact
{
    protected $dates = [
        'met_at',
    ];
}
```

Standardmäßig werden die Spalten `created_at` und `updated_at` zu Carbon-Instanzen mutiert. Gleiches gilt für alle Spalten, die im Array `$dates` stehen.

Das ist etwas einfacher und übersichtlicher, also Zeitstempel- oder andere Datumsspalten explizit in `$casts` aufzuführen. Aber letztlich ist das Geschmackssache.

Eloquent-Collections

Wenn Sie eine Eloquent-Abfrage durchführen, bei der potenziell mehrere Zeilen zurückzugeben werden können, wird das Ergebnis nicht als Array übergeben, sondern »verpackt« in einer Eloquent-Collection, einer spezielleren Variante einer Collection. Werfen wir einen Blick auf Collections und Eloquent-Collections und was sie gegenüber einfachen Arrays auszeichnet.

Eine kurze Einführung in Collections

Laravels `Collection`-Objekte (`Illuminate\Support\Collection`) könnte man als Arrays auf Steroiden bezeichnen. Die Methoden, die sie mitbringen, sind so hilfreich, dass Sie sie nach einer Weile wahrscheinlich auch in Nicht-Laravel-Projekten vermissen werden – Abhilfe schafft dann das Paket Tightenco/Collect (*https://github.com/tightenco/collect*).

Am einfachsten erstellt man eine Collection mit dem `collect()`-Helfer. Übergeben Sie entweder ein Array oder verwenden Sie `collect()` ohne Argumente, damit eine leere Sammlung erstellt wird, deren Elemente Sie erst später festlegen. Probieren wir es einmal aus:

```
$collection = collect([1, 2, 3]);
```

Nehmen wir an, wir wollen keine geraden Zahlen in unserer Collection haben:

```
$odds = $collection->reject(function ($item) {
    return $item % 2 === 0;
});
```

Vielleicht brauchen wir eine Version der Collection, bei der jeder Artikel mit 10 multipliziert wird? Das können wir wie folgt erreichen:

```
$multiplied = $collection->map(function ($item) {
    return $item * 10;
});
```

Wir könnten die Collection auch auf gerade Zahlen beschränken, diese mit 10 multiplizieren und mit `sum()` auf eine einzige Zahl reduzieren:

```
$sum = $collection
    ->filter(function ($item) {
        return $item % 2 == 0;
    })->map(function ($item) {
        return $item * 10;
    })->sum();
```

Wie Sie sehen, stellen Collections eine Reihe von Methoden zur Verfügung, die optional verkettet werden können, um funktionale Operationen auf den enthaltenen Werten durchzuführen. Sie bieten die gleiche Funktionalität von nativen PHP-Methoden wie `array_map()` und `array_reduce()`, aber Sie müssen sich nicht die bei den PHP-Varianten unvorhersehbare Reihenfolge der Parameter merken. Zudem ist die Syntax mit Methodenverkettung erheblich besser lesbar.

Es gibt mehr als 60 Methoden in der Klasse `Collection`, darunter solche wie `max()`, `whereIn()`, `flatten()` und `flip()`, aber leider würde es den Rahmen dieses Buchs sprengen, sie alle vorzustellen. Auf einige dieser Methoden werden wir in Kapitel 17 eingehen. Alle Methoden sind im Abschnitt Collections (*https://laravel.com/docs/master/collections*) der Laravel-Dokumentation beschrieben.

Collection statt Arrays

Collections können auch in jedem anderen Kontext anstelle von Arrays verwendet werden (außer beim Typehinting); sie erlauben Iterationen mit `foreach` und, sofern sie Schlüssel besitzen, den bekannten Array-Zugriff, also beispielsweise Zuweisungen wie `$a = $collection['a']`.

Der Zusatznutzen von Eloquent-Collections

Eine Eloquent-Collection erweitert eine normale Collection um zusätzliche Möglichkeiten zum einfacheren Umgang mit Abfrageergebnissen.

Leider reicht der Platz nicht, um all diese Ergänzungen im Einzelnen zu beschreiben, aber für alle gilt, dass sie sich auf die Interaktion mit Collections konzentrieren, die eine ganz spezielle Form von Objekten enthalten, nämlich die Repräsentationen von Datenbankzeilen.

Eloquent-Collection besitzen beispielsweise eine Methode namens `modelKeys()`, die ein Array mit den Primärschlüsseln der in der Collection enthaltenen Instanzen zurückgibt. `find($id)` wiederum sucht nach der Instanz, die einen Primärschlüssel `$id` besitzt.

Zudem gibt es die Möglichkeit, für ein bestimmtes Modell festzulegen, dass es seine Ergebnisse in einer benutzerdefinierten Collection-Klasse zurückgeben soll. Wenn Sie beispielsweise bei einem `Order`-Modell zusätzliche, spezifische Methoden hinzufügen möchten, etwa um die finanziellen Details der Bestellungen zusammenzufassen, könnten Sie eine Klasse `OrderCollection` erstellen, die `Illuminate\Database\Eloquent\Collection` erweitert, und diese in Ihrem Modell registrieren, wie Beispiel 5-40 zeigt.

Beispiel 5-40: Benutzerdefinierte Collection-Klassen für Eloquent-Modelle

```
...
class OrderCollection extends Collection
{
    public function sumBillableAmount()
    {
        return $this->reduce(function ($carry, $order) {
            return $carry + ($order->billable ? $order->amount : 0);
        }, 0);
    }
}

...
class Order extends Model
{
    public function newCollection(array $models = [])
    {
        return new OrderCollection($models);
    }
```

Jedes Mal, wenn Sie eine Collection von `Orders` – z. B. von `Order::all()` – zurückgeliefert bekommen, wird das Ergebnis nun eine Instanz der Klasse `OrderCollection` sein:

```
$orders = Order::all();
$billableAmount = $orders->sumBillableAmount();
```

Serialisierung mit Eloquent

Als Serialisierung bezeichnet man es, wenn man eine komplexere Struktur nimmt – ein Array oder ein Objekt – und es in eine Zeichenkette umwandelt. In einem webbasierten Kontext wird diese Zeichenkette meist im JSON-Format benutzt, aber sie könnte prinzipiell auch andere Formen aufweisen.

Die Serialisierung komplexer Datenbankeinträge kann, naja, eben kompliziert sein, und bei dieser Aufgabe tun sich viele ORMs schwer. Glücklicherweise gibt es bei Eloquent zwei leistungsstarke Methoden: `toArray()` und `toJson()`. Auf Collections kann man `toArray()` und `toJson()` natürlich genauso wie auf einzelne Instanzen anwenden, also wären alle folgenden Zuweisungen möglich:

```
$contactArray = Contact::first()->toArray();
$contactJson = Contact::first()->toJson();
```

```
$contactsArray = Contact::all()->toArray();
$contactsJson = Contact::all()->toJson();
```

Sie können eine Eloquent-Instanz oder -Collection zwar auch explizit z. B. mit `$string = (string) $contact;` in eine Zeichenkette umwandeln, aber sowohl Modelle als auch Collections werden dabei im Hintergrund einfach `toJson()` ausführen und das Ergebnis liefern.

Modelle direkt mit Routenmethoden zurückgeben

Laravels Router konvertiert alles, was Routen zurückgeben, in eine Zeichenkette, sodass man einen cleveren Trick anwenden kann, indem man das Ergebnis eines Eloquent-Aufrufs an einen Controller zurückgibt: Es wird automatisch in eine Zeichenkette und damit in JSON umgewandelt. Eine Route, die JSON zurückgeben soll, lässt sich also so einfach definieren wie in Beispiel 5-41.

Beispiel 5-41: JSON aus einer Route zurückgeben

```
// routes/web.php
Route::get('api/contacts', function () {
    return Contact::all();
});

Route::get('api/contacts/{id}', function ($id) {
    return Contact::findOrFail($id);
});
```

Attribute aus JSON ausblenden

Rückgaben im JSON-Format werden sehr häufig im Zusammenspiel mit APIs verwendet, und oft will man in diesem Kontext bestimmte Attribute ausblenden. Mit Eloquent lässt sich das einfach bewerkstelligen.

Sie können Attribute entweder auf eine schwarze Liste namens `$hidden` setzen und dadurch ausblenden ...

```
class Contact extends Model
{
    public $hidden = ['password', 'remember_token'];
```

... oder Attribute einer Whitelist namens `$visible` hinzufügen, um nur die darin enthaltenen Attribute anzuzeigen:

```
class Contact extends Model
{
    public $visible = ['name', 'email', 'status'];
```

Das funktioniert auch für Beziehungen zwischen Relationen, beispielsweise für alle Kontakte eines Benutzers:

```
class User extends Model
{
    public $hidden = ['contacts'];
```

```
public function contacts()
{
    return $this->hasMany(Contact::class);
}
```

Daten einer Beziehung erhalten

Wenn eine Datenbankzeile eingelesen wird, gilt das standardmäßig natürlich nicht für die über eine Beziehung verknüpften Daten, sodass es erst einmal egal ist, ob Sie diese Beziehung verstecken oder nicht. Wie Sie gleich sehen werden, ist es aber auch möglich, einen Datensatz *mit* all seinen verknüpften Elementen abzufragen. Falls Sie eine Beziehung als »verborgen« kennzeichnen, werden die verknüpften Elemente auch in diesem Kontext nicht in der serialisierten Version dieses Datensatzes enthalten sein.

Mit dem folgenden Aufruf – ein weiterer kleiner Vorgriff – können Sie einen `User` mit allen Kontakten abfragen – vorausgesetzt, Sie haben die Beziehung korrekt eingerichtet:

```
$user = User::with('contacts')->first();
```

Es kann vorkommen, dass Sie ein Attribut nur für einen einzigen Aufruf sichtbar machen möchten. Dafür gibt es die Eloquent-Methode `makeVisible()`:

```
$array = $user->makeVisible('remember_token')->toArray();
```

Hinzufügen einer generierten Spalte zu Array- und JSON-Ausgaben

Falls Sie einen Akzessor für eine Spalte erstellt haben, die nicht existiert – zum Beispiel unsere Spalte `full_name` aus Beispiel 5-35 –, können Sie sie mit `$appends` den Array- und JSON-Ausgaben des Modells hinzufügen:

```
class Contact extends Model
{
    protected $appends = ['full_name'];

    public function getFullNameAttribute()
    {
        return "{$this->first_name} {$this->last_name}";
    }
}
```

Beziehungen mit Eloquent

In einem relationalen Datenbankmodell arbeitet man mit Tabellen, die untereinander *verwandt* sind bzw. *in Beziehung* miteinander stehen, weil sie anhand bestimmter Eigenschaften miteinander verknüpft sind. In diesem Zusammenhang werden Tabellen auch als *Relationen* bezeichnet – daher der Name des Datenbankmodells: *relational*. Eloquent bietet einfache und leistungsstarke Werkzeuge, um diese Beziehungen zwischen Tabellen festzulegen.

In vielen bisherigen Beispielen in diesem Kapitel ging es um Benutzer, die viele Kontakte haben können – eine ziemlich häufig vorkommende Situation.

In relationalen Datenbanken und einem ORM wie Eloquent bezeichnet man das als *1:n-*, *Eins-zu-viele-* oder *One-to-many*-Beziehung: Ein Benutzer *hat viele* Kontakte (*has many* `Contacts`).

In einem CRM könnte es auch vorkommen, dass ein einzelner Kontakt vielen Benutzern zugeordnet wird. Dann hätte man eine *Viele-zu-viele-* bzw. *Many-to-many*-Beziehung: Viele Benutzer können einem Kontakt zugeordnet und gleichzeitig kann jeder Benutzer mit vielen Kontakten verbunden werden. Ein Benutzer *hat viele* (*has many*) und *gehört zu* (*belongs to*) vielen Kontakten.

Besäße jeder Kontakt mehrere Telefonnummern, könnte man zudem sagen, dass ein Benutzer viele Telefonnummern *über* bzw. *durch* seine Kontakte hat. In diesem Fall wäre der Kontakt eine Art »Vermittler« zwischen dem Benutzer und den Telefonnummern.

Nun hat sicherlich auch jeder Kontakt eine Adresse. Obwohl man die benötigten Adressfelder auch alle direkt beim Kontakt speichern könnte, ließe sich aber alternativ auch ein eigenes Modell `Adresse` einrichten – dann könnte man sagen, der Kontakt *hat eine* Adresse (*has one*).

Als letztes Gedankenexperiment stellen wir uns noch vor, wir wollten in unserer Anwendung den Benutzern erlauben, bestimmte Einträge als Favoriten zu kennzeichnen, beispielsweise deren wichtigste Kontakte, aber zusätzlich auch noch eine ganz andere Kategorie, zum Beispiel Events. Das wäre dann eine *polymorphe* Beziehung, in der ein Benutzer viele Favoriten haben könnte, von denen aber einige Kontakt- und andere Ereignis-Favoriten wären.

Jetzt wollen wir uns anschauen, wie man all diese unterschiedlichen Beziehungen definiert und nutzt.

1:1-Beziehungen

Beginnen wir einfach: Ein Kontakt hat eine Telefonnummer. Diese Beziehung ist definiert in Beispiel 5-42.

Beispiel 5-42: Definition einer 1:1-Beziehung

```
class Contact extends Model
{
    public function phoneNumber()
    {
        return $this->hasOne(PhoneNumber::class);
    }
```

Die Methoden, mit denen Beziehungen definiert werden, stehen im Modell selbst (`$this->hasOne()`) und erwarten den voll qualifizierten Namen der Klasse, die Ziel der Beziehung sein soll (hier `PhoneNumber`).

Wie sollte diese Beziehung in der Datenbank definiert sein? Da wir festgelegt haben, dass ein Kontakt eine Telefonnummer hat, erwartet Eloquent, dass die Tabelle, die zur Klasse `PhoneNumber` gehört (vermutlich `phone_numbers`), eine Spalte

contact_id enthält. Wenn Sie einen anderen Namen verwenden (z.B. owner_id), müssen Sie die Definition entsprechend ändern:

```
return $this->hasOne(PhoneNumber::class, 'owner_id');
```

Und so greifen wir auf die Telefonnummer eines Kontakts zu:

```
$contact = Contact::first();
$contactPhone = $contact->phoneNumber;
```

Beachten Sie bitte, dass wir die Methode in Beispiel 5-42 mit phoneNumber() definieren, aber sie mit ->phoneNumber aufrufen. Magisch! Sie könnten auch mit ->phone_number darauf zugreifen. Beides gibt eine vollständige Instanz des zugehörigen Telefonnummer-Eintrags zurück.

Wie können wir über die Telefonnummer an den Kontakt kommen? Auch dafür gibt es eine Methode (siehe Beispiel 5-43).

Beispiel 5-43: Umkehrung einer 1:1-Beziehung definieren

```
class PhoneNumber extends Model
{
    public function contact()
    {
        return $this->belongsTo(Contact::class);
    }
```

Dann greifen wir auf analoge Weise darauf zu:

```
$contact = $phoneNumber->contact;
```

Verknüpfte Daten speichern

Jeder Beziehungstyp hat seine Eigenarten, was die Verknüpfung von Modellen betrifft, aber im Wesentlichen funktioniert es so: Man übergibt eine einzelne Instanz an save() oder ein Array von Instanzen an saveMany(). Oder man übergibt Eigenschaften an create() oder createMany(), sodass direkt entsprechende neue Instanzen erstellt und gespeichert werden:

```
$contact = Contact::first();

$phoneNumber = new PhoneNumber;
$phoneNumber->number = 8008675309;
$contact->phoneNumbers()->save($phoneNumber);

// oder

$contact->phoneNumbers()->saveMany([
    PhoneNumber::find(1),
    PhoneNumber::find(2),
]);

// oder

$contact->phoneNumbers()->create([
    'number' => '+13138675309',
```

```
]);

// oder

$contact->phoneNumbers()->createMany([
    ['number' => '+13138675309'],
    ['number' => '+15556060842'],
]);
```

Die Methode `createMany()` ist erst ab Laravel 5.4 verfügbar.

1:n-Beziehung

Die 1:n-Beziehung kommt am häufigsten vor. Schauen wir uns an, wie man definiert, dass ein Benutzer viele Kontakte haben kann (Beispiel 5-44).

Beispiel 5-44: Definition einer 1:n-Beziehung

```
class User extends Model
{
    public function contacts()
    {
        return $this->hasMany(Contact::class);
    }
```

Wieder wird erwartet, dass die zugrunde liegenden Tabelle des `Contact`-Modells (vermutlich `contacts`) eine Spalte `user_id` enthält. Falls nicht, übergeben Sie den korrekten Spaltennamen als zweiten Parameter von `hasMany()`.

Die Kontakte eines Benutzers erhalten wir so:

```
$user = User::first();
$usersContacts = $user->contacts;
```

Genau wie bei 1:1-Beziehungen verwenden wir den Namen der Beziehungsmethode und benutzen sie, als wäre sie eine Eigenschaft und nicht eine Methode. `hasMany()` gibt jedoch anstelle einer einzelnen Modellinstanz eine Collection zurück. Und zwar eine Eloquent-Collection, sodass man damit viele schöne Dinge anstellen kann, z. B. Kontakte herausfiltern, die als Spender (*donor*) gekennzeichnet sind, oder die Gesamtsumme aller Bestellungen eines Kontakts berechnen:

```
$donors = $user->contacts->filter(function ($contact) {
    return $contact->status == 'donor';
});

$lifetimeValue = $contact->orders->reduce(function ($carry, $order) {
    return $carry + $order->amount;
}, 0);
```

Genau wie 1:1-Beziehungen können wir auch die Umkehrung der Beziehung definieren (Beispiel 5-45).

Beispiel 5-45: Umkehrung einer 1:n-Beziehung definieren

```
class Contact extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
```

Und genau wie bei 1:1-Beziehungen können wir über den Kontakt auf den Benutzer zugreifen:

```
$userName = $contact->user->name;
```

Assoziieren und Deassoziieren verknüpfter Elemente

Meistens fügen wir einem Datenbankeintrag ein neues verknüpftes Element hinzu, indem wir `save()` auf dem Elternteil ausführen und der Methode gleichzeitig das dazugehörige Element übergeben wie in `$user->contacts()->save($contact)`. Wenn Sie direkt auf dem verknüpften (»untergeordneten«) Eintrag arbeiten möchten, können Sie `associate()` bzw. `dissociate()` an die Methode anhängen, die die `belongsTo`-Beziehung zurückgibt:

```
$contact = Contact::first();

$contact->user()->associate(User::first());
$contact->save();

// und später

$contact->user()->dissociate();
$contact->save();
```

Beziehungen als Query Builder verwenden. Bisher haben wir den Methodennamen – z. B. `contacts()` – genommen und ihn wie eine Eigenschaft – z. B. `$user->contacts` – aufgerufen. Was passiert, wenn wir tatsächlich die Methode aufrufen? Anstatt direkt auf der Beziehung zu arbeiten, wird dann ein Query-Builder-Objekt mit voreingestelltem Geltungsbereich zurückgegeben.

Wäre also `$user` die Instanz eines bestimmten Benutzer, etwa mit der ID 1, und würden Sie darauf die Methode `contacts()` anwenden, erhielten Sie einen Query Builder, der bereits auf »alle Kontakte, die ein Feld `user_id` mit dem Wert `1` enthalten« eingeschränkt wäre. Darauf aufbauend ließe sich dann eine funktionale Abfrage konstruieren:

```
$donors = $user->contacts()->where('status', 'donor')->get();
```

Nur Datensätze auswählen, die ein Bezugselement enthalten. Sie können die Abfrage mit `has()` auf Datensätze beschränken, die bestimmte Kriterien in Bezug auf ihre zugehörigen Elemente erfüllen:

```
$postsWithComments = Post::has('comments')->get();
```

Sie können die Kriterien auch enger formulieren:

```
$postsWithManyComments = Post::has('comments', '>=', 5)->get();
```

Oder per Punktnotation verschachteln:

```
$usersWithPhoneBooks = User::has('contacts.phoneNumbers')->get();
```

Und schließlich können Sie benutzerdefinierte Abfragen für die zugehörigen Elemente schreiben:

```
// Ruft alle Kontakte mit einer Telefonnummer ab, die die Zeichenkette "867-5309"
// enthalten
$jennyIGotYourNumber = Contact::whereHas('phoneNumbers', function ($query) {
    $query->where('number', 'like', '%867-5309%');
});
```

Vermittelte 1:n-Beziehung

`hasManyThrough()` ist wirklich eine praktische Methode, um Beziehungen einer Beziehung einzubinden. Denken Sie an das frühere Beispiel, in dem ein Benutzer viele Kontakte hat und jeder Kontakt viele Telefonnummern. Wie kommen Sie an eine Liste aller Telefonnummern eines Benutzers? Dazu bedarf es einer vermittelten 1:n-Beziehung oder anders ausgedrückt: einer Eins-zu-viele-durch- oder Has-Many-Through-Beziehung.

Diese Struktur geht davon aus, dass Ihre Tabelle `contacts` eine `user_id` besitzt, um die Kontakte mit den Benutzern zu verknüpfen, und die Tabelle `phone_numbers` eine `contact_id`, um sie den Kontakten zuzuordnen. Dann definieren wir die Beziehung im `User`-Modell wie in Beispiel 5-46.

Beispiel 5-46: Definition einer Has-Many-Through-Beziehung

```
class User extends Model
{
    public function phoneNumbers()
    {
        return $this->hasManyThrough(PhoneNumber::class, Contact::class);
    }
```

Sie würden auf diese Beziehung mit `$user->phone_numbers` zugreifen, und wie immer können Sie den Beziehungsschlüssel des vermittelnden bzw. zwischengeschalteten Modells (mit dem dritten Parameter von `hasManyThrough()`) und den Beziehungsschlüssel des entfernten Ziel-Modells (mit dem vierten Parameter) anpassen.

Vermitttelte 1:1-Beziehung

`hasOneThrough()` ähnelt `hasManyThrough()`, aber anstatt über einen »Vermittler« auf *viele* verwandte Elemente zuzugreifen, geht es hier um ein *einzelnes* verwandtes Element, das über ein *einzelnes* Zwischenelement erreicht wird.

Stellen Sie sich Folgendes vor: Jeder Benutzer arbeitet in einer bestimmten Firma; diese Firma besitzt eine einzige Telefonnummer; und Sie möchten über den Benutzer an die Telefonnummer der Firma kommen. Dann braucht man eine vermittelte 1:1-Beziehung oder mit anderen Worten: eine Eins-zu-eins-durch- bzw. Has-One-Through-Beziehung.

Beispiel 5-47: Definition einer Has-One-Through-Beziehung

```
class User extends Model
{
    public function phoneNumber()
    {
        return $this->hasOneThrough(PhoneNumber::class, Company::class);
    }
```

m:n-Beziehung

Jetzt fängt es an, komplizierter zu werden. Bleiben wir bei unserem Beispiel eines CRM, das es einem Benutzer ermöglicht, viele Kontakte zu haben, und bei dem jeder Kontakt mit mehreren Benutzern verknüpft ist.

Zuerst definieren wir dazu die Beziehung im `User`-Modell, siehe Beispiel 5-48.

Beispiel 5-48: Definition einer m:n-Beziehung

```
class User extends Model
{
    public function contacts()
    {
        return $this->belongsToMany(Contact::class);
    }
}
```

Da dies eine Viele-zu-viele-Beziehung ist, sieht die Umkehrung genauso aus (Beispiel 5-49).

Beispiel 5-49: Umkehrung einer m:n-Beziehung definieren

```
class Contact extends Model
{
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

Bei einer m:n-Beziehung kann ein `Contact` keine einzelne `user_id`-Spalte mehr haben, da er ja mehreren Benutzern zugeordnet werden kann, und dasselbe gilt umgekehrt für einen einzelnen `User` und eine jetzt dysfunktional gewordene einzelne `contact_id`-Spalte. Deshalb wird bei Viele-zu-viele-Beziehungen eine Verbindungstabelle zur Kopplung der Datensätze verwendet. Die gebräuchliche Konvention zur Benennung dieser Tabelle lautet, dass die beiden einzelnen Tabellennamen singu-

larisiert, alphabetisch geordnet und durch einen Unterstrich zusammengefügt werden.

Da wir `users` und `contacts` verknüpfen, sollte unsere Verbindungstabelle also den Namen `contact_user` erhalten (wenn Sie den Tabellennamen anpassen möchten, übergeben Sie ihn als zweiten Parameter an die Methode `belongsToMany()`). Sie muss zwei Spalten enthalten: `contact_id` und `user_id`.

Genau wie bei `hasMany()` erhalten wir nach der Definition entsprechender Methoden im Modell Zugriff auf Collections der zugehörigen Elemente, und zwar von beiden Seiten (Beispiel 5-50).

Beispiel 5-50: Zugriff auf die verknüpften Elemente von beiden Seiten einer m:n-Beziehung

```
$user = User::first();

$user->contacts->each(function ($contact) {
    // Hier Anweisungen ...
});

$contact = Contact::first();

$contact->users->each(function ($user) {
    // Hier Anweisungen ...
});

$donors = $user->contacts()->where('status', 'donor')->get();
```

Daten aus der Verbindungstabelle abrufen. Das Spezielle an einer m:n-Beziehung ist die Tatsache, dass eine Verbindungstabelle benötigt wird. Je weniger Spalten die Verbindungstabelle hat, desto besser, aber es gibt einige Fälle, in denen es dennoch sinnvoll sein kann, über die beiden Fremdschlüssel-Spalten hinaus darin weitere Informationen zu speichern – zum Beispiel, wenn Sie ein Feld `created_at` benutzen möchten, um festzuhalten, wann eine Verknüpfung zwischen zwei Zeilen erstellt wurde.

Um zusätzliche Felder zu speichern, müssen Sie sie, wie in Beispiel 5-51 gezeigt, in die Beziehungsdefinition aufnehmen. Sie können zusätzliche Spalten mit `withPivot()` definieren oder die beiden Zeitstempel `created_at` und `updated_at` mit `withTimestamps()` hinzufügen.

Beispiel 5-51: Hinzufügen von Spalten zu einer Verbindungstabelle

```
public function contacts()
{
    return $this->belongsToMany(Contact::class)
        ->withTimestamps()
        ->withPivot('status', 'preferred_greeting');
}
```

Wenn Sie jetzt über eine Beziehungsmethode Modellinstanzen abfragen, besitzen diese eine Eigenschaft namens `pivot`, die ihren jeweiligen Platz in der Verbindungs-

tabelle darstellt. (Die Verbindungstabelle wird bei Laravel im Original als Pivot-Tabelle bezeichnet. Da damit aber in der Regel spezielle Tabellen zur Aggregierung von Daten bezeichnet werden, verwenden wir in der deutschen Übersetzung den neutralen Begriff der Verbindungstabelle.) Sie können also so etwas wie Beispiel 5-52 machen.

Beispiel 5-52: Weitere Daten aus einem Eintrag in der Verbindungstabelle abrufen

```
$user = User::first();

$user->contacts->each(function ($contact) {
    echo sprintf(
        'Contact associated with this user at: %s',
        $contact->pivot->created_at
    );
});
```

Die Eigenschaft `pivot` können Sie nach Belieben umbenennen, indem Sie die Methode `as()` verwenden, wie Beispiel 5-53 zeigt.

Beispiel 5-53: Umbenennen der Eigenschaft pivot

```
// User-Modell
public function groups()
{
    return $this->belongsToMany(Group::class)
        ->withTimestamps()
        ->as('membership');
}

// Verwendung dieser Beziehung:
User::first()->groups->each(function ($group) {
    echo sprintf(
        'User joined this group at: %s',
        $group->membership->created_at
    );
});
```

Besondere Aspekte beim Verbinden und Lösen verknüpfter Elemente bei m:n-Beziehungen

Da eine Verbindungstabelle eigene, zusätzliche Eigenschaften neben den beiden Fremdschlüsseln aufweisen kann, muss man diese Eigenschaften auch festlegen können, wenn eine Kopplung hergestellt wird (also eine neue Zeile in die Verbindungstabelle eingetragen bzw. eine vorhandene aktualisiert wird). Sie können das erreichen, indem Sie ein Array als zweiten Parameter an `save()` übergeben:

```
$user = User::first();
$contact = Contact::first();
$user->contacts()->save($contact, ['status' => 'donor']);
```

Außerdem können Sie die Methoden `attach()` und `detach()` verwenden, wobei nicht eine ganze Instanz eines verwandten Elements übergeben wird, sondern nur eine ID. Die beiden Methoden funktionieren prinzipiell genauso wie `save()`, akzeptieren aber auch ein Array mit mehreren IDs:

```
$user = User::first();
$user->contacts()->attach(1);
$user->contacts()->attach(2, ['status' => 'donor']);
$user->contacts()->attach([1, 2, 3]);
$user->contacts()->attach([
    1 => ['status' => 'donor'],
    2,
    3,
]);

$user->contacts()->detach(1);
$user->contacts()->detach([1, 2]);
$user->contacts()->detach(); // Alle assoziierten Kontakte "lösen"
```

Anstatt die Verbindungen zwischen Datensätzen mit `attach()` und `detach()` zu steuern, können Sie mit der Methode `toggle()` auch einfach zwischen den beiden möglichen Zuständen umschalten: Mit `toggle()` wird für jede angegebene ID geprüft, ob sie gerade verknüpft ist oder nicht – ist die ID verknüpft, wird die Verbindung gelöst; ist sie nicht verknüpft, wird die Verbindung hergestellt:

```
$user->contacts()->toggle([1, 2, 3]);
```

Sie können auch `updateExistingPivot()` verwenden, um Änderungen nur an Eigenschaften in der Verbindungstabelle vorzunehmen, übergeben wieder in einem Array:

```
$user->contacts()->updateExistingPivot($contactId, [
    'status' => 'inactive',
]);
```

Und falls Sie *alle* aktuellen Verbindungen durch neue ersetzen möchten – also alle vorhandenen löschen und neue definieren –, können Sie ein Array an `sync()` übergeben:

```
$user->contacts()->sync([1, 2, 3]);
$user->contacts()->sync([
    1 => ['status' => 'donor'],
    2,
    3,
]);
```

Polymorphe Beziehungen

In einer polymorphen Beziehung gibt es mehrere Modelle, die den gleichen Beziehungstyp aufweisen. Wir werden für ein System zur Markierung von Favoriten ein Modell namens `Stars` verwenden, um mit diesen »Sternen« die Favoriten zu kennzeichnen. Dabei soll ein Benutzer sowohl Kontakte als auch Events mit einem Stern versehen können, und damit klärt sich auch die Bezeichnung *polymorph*, also *viel-*

gestaltig: Es gibt eine gemeinsame Schnittstelle für Objekte unterschiedlicher Gestalt.

Wir brauchen dazu insgesamt drei Tabellen und Modelle: `Star`, `Contact` und `Event` (eigentlich vier, technisch gesprochen, weil wir auch die Tabelle `user` und das Modell `User` benötigen, aber dazu kommen wir noch). Die Tabellen `contacts` und `events` sind normale Tabellen ohne große Besonderheiten, die entscheidende Tabelle `stars` wird die Spalten `id`, `starrable_id` und `starrable_type` enthalten. Für jeden `Star` legen wir fest, welche Objektkategorie bzw. welchen »Typ« (`Contact` oder `Event`) er auszeichnet und welche ID das mit einem Stern ausgezeichnete Objekt in seiner »Heimattabelle« hat.

Lassen Sie uns die Modelle erstellen so wie in Beispiel 5-54.

Beispiel 5-54: Modelle für ein polymorphes Favoritensystem erstellen

```
class Star extends Model
{
    public function starrable()
    {
        return $this->morphTo();
    }
}

class Contact extends Model
{
    public function stars()
    {
        return $this->morphMany(Star::class, 'starrable');
    }
}

class Event extends Model
{
    public function stars()
    {
        return $this->morphMany(Star::class, 'starrable');
    }
}
```

Aber wie erzeugen wir jetzt einen Stern als Markierung eines Favoriten?

```
$contact = Contact::first();
$contact->stars()->create();
```

So einfach geht das: Der Kontakt ist nun – zumindest datenbanktechnisch – mit einem Sternchen versehen.

Um alle `Stars` zu finden, die einen bestimmten `Contact` markieren, rufen wir die Methode `stars()` auf wie in Beispiel 5-55.

Beispiel 5-55: Instanzen einer polymorphen Beziehung abrufen

```
$contact = Contact::first();
```

```
$contact->stars->each(function ($star) {
    // Anweisungen hier ...
});
```

Wenn wir eine Instanz von `Star` haben, können wir ihr »Ziel« abfragen (also die Instanz, die als Favorit markiert wurde), indem wir die Methode aufrufen, mit der wir ihre `morphTo`-Beziehung definiert haben, in diesem Fall `starrable()`. Werfen Sie dazu einen Blick auf Beispiel 5-56.

Beispiel 5-56: Das Ziel einer polymorphen Instanz abrufen

```
$stars = Star::all();

$stars->each(function ($star) {
    var_dump($star->starrable); // Eine Instanz von Contact oder Event
});
```

Vielleicht fragen Sie sich: »Wie kann ich herausfinden, wer einen Favoriten-Stern gesetzt hat?« Eine gute Frage! Aber die Lösung ist zum Glück einfach: Die Tabelle `stars` muss um eine Spalte `user_id` erweitert werden; und es muss festgelegt werden, dass a) ein `User` *viele* `Stars` setzen kann und b) ein `Star` zu *genau einem* `User` gehört – eine 1:n-Beziehung (Beispiel 5-57). Die Tabelle `stars` wird damit quasi zu einer Verbindungstabelle zwischen Benutzern und Kontakten bzw. Events.

Beispiel 5-57: Erweiterung eines polymorphen Systems zur Differenzierung nach Benutzern

```
class Star extends Model
{
    public function starrable()
    {
        return $this->morphsTo;
    }

    public function user()
    {
        return $this->belongsTo(User::class);
    }
}

class User extends Model
{
    public function stars()
    {
        return $this->hasMany(Star::class);
    }
}
```

Das war's! Sie können nun `$star->user` oder `$user->stars` ausführen, um anhand eines Sterns den Benutzer zu finden, der diesen vergeben, oder um eine Liste der Favoritensterne zu bekommen, die ein Benutzer insgesamt verteilt hat. Jetzt müssen Sie allerdings auch den `User` festlegen, wenn ein neuer Stern vergeben wird:

```
$user = User::first();
$event = Event::first();
$event->stars()->create(['user_id' => $user->id]);
```

Polymorphe m:n-Beziehung

Die komplexeste und am wenigsten gebräuchliche Variante von Beziehungen ist die polymorphe Viele-zu-viele-Beziehung.

Das häufigste Beispiel für diesen Beziehungstyp wäre die Verwendung von Schlagwörtern (*tags*). Und um auf der sicheren Seite zu sein, werde ich dieses Beispiel auch hier benutzen. Stellen Sie sich also vor, Sie würden gerne sowohl Kontakte wie Events verschlagworten. Das Besondere bei einem m:n-Polymorphismus besteht einerseits darin, dass es eine Viele-zu-viele-Beziehung ist: Jedes Schlagwort bzw. jeder Tag kann auf mehrere Elemente angewendet werden, und jedes Element kann mehrere Schlagwörter haben. Aber daneben ist diese Beziehung eben auch noch polymorph: Tags können mit Elementen verschiedenen Typs verknüpft werden, so ähnlich wie im letzten Beispiel, bei dem wir Favoriten benutzt haben. Was die Datenbank betrifft, bauen wir wieder die Struktur einer polymorphen Beziehung auf (wie bei den Favoriten), fügen aber noch eine Verbindungstabelle hinzu.

Das bedeutet, dass wir drei Standardtabellen, `contacts`, `events` und `tags` brauchen, die alle jeweils eine ID und die gewünschten Eigenschaften enthalten, *und* eine zusätzliche Tabelle `taggables`, mit den Spalten `tag_id`, `taggable_id` und `taggable_type`. Jeder Eintrag in der Tabelle `taggables` verknüpft ein Tag mit einem Typus von Taggable-Inhalten.

Lassen Sie uns jetzt die Modelle für diese Beziehung definieren so wie in Beispiel 5-58.

Beispiel 5-58: Definition einer polymorphen m:n-Beziehung

```
class Contact extends Model
{
    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}

class Event extends Model
{
    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}

class Tag extends Model
{
    public function contacts()
    {
```

```
        return $this->morphedByMany(Contact::class, 'taggable');
    }

    public function events()
    {
        return $this->morphedByMany(Event::class, 'taggable');
    }
}
```

Und so erstellen Sie Ihr erstes Tag:

```
$tag = Tag::firstOrCreate(['name' => 'likes-cheese']);
$contact = Contact::first();
$contact->tags()->attach($tag->id);
```

Ergebnisse für diese Beziehung bekommen wir wie gewohnt, siehe Beispiel 5-59.

Beispiel 5-59: Zugriff auf die verknüpften Elemente von beiden Seiten einer polymorphen m:n-Beziehung

```
$contact = Contact::first();

$contact->tags->each(function ($tag) {
    // Anweisungen hier ...
});

$tag = Tag::first();
$tag->contacts->each(function ($contact) {
    // Anweisungen hier ...
});
```

Aktualisierung von Zeitstempeln durch verknüpfte Datensätze

Wie Sie wissen, haben alle Modelle standardmäßig die Zeitstempel-Spalten `created_at` und `updated_at`. Eloquent aktualisiert den Zeitstempel `updated_at` automatisch, wenn Sie Änderungen an einem Datensatz vornehmen.

Wenn ein verknüpftes Element eine `belongsTo`- oder `belongsToMany`-Beziehung zu einem »entfernten« Element hat, kann es durchaus sinnvoll sein, das (logisch meist übergeordnete) entfernte Element auch bei jeder Veränderung des verknüpften Elements als aktualisiert zu markieren. Wenn beispielsweise eine Telefonnummer aktualisiert wird, sollte vielleicht auch der Kontakt, zu dem sie gehört, als aktualisiert gekennzeichnet werden.

Erreichen können wir das, indem wir der Definition des verknüpften Modells eine als Array notierte Eigenschaft `$touches` hinzufügen wie in Beispiel 5-60.

Beispiel 5-60: Aktualisieren eines übergeordneten Datensatzes, wenn der untergeordnete Datensatz aktualisiert wird

```
class PhoneNumber extends Model
{
    protected $touches = ['contact'];
```

```
    public function contact()
    {
        return $this->belongsTo(Contact::class);
    }
}
```

Eager Loading

Standardmäßig lädt Eloquent die Daten aus Beziehungen mit einem Verfahren namens *Lazy Loading*. Das bedeutet, dass beim Laden einer Modellinstanz die abhängigen Modelle nicht direkt mitgeladen werden. Vielmehr werden sie erst dann nachgeladen, wenn auf sie auch tatsächlich zugegriffen wird; Laravel verhält sich beim Ladevorgang quasi »faul« (»lazy«) und spart sich die Arbeit, bis es absolut nötig ist.

Das kann zu einem Performance-Problem werden, wenn Sie über eine große Menge von Datensätzen iterieren und für jeden dieser Datensätze verknüpfte Elemente existieren. Die Krux beim »faulen« Laden: Es kann die Zahl der Datenbankabfragen signifikant erhöhen (oft als *N*+1-Problem bezeichnet). Laravel führt beispielsweise jedes Mal, wenn die Schleife in Beispiel 5-61 durchlaufen wird, eine neue Datenbankabfrage aus, um die Telefonnummern für den aktuellen Kontakt zu suchen.

Beispiel 5-61: Abrufen eines verknüpften Elements für jedes Element einer Liste

```
$contacts = Contact::all();

foreach ($contacts as $contact) {
    foreach ($contact->phone_numbers as $phone_number) {
        echo $phone_number->number;
    }
}
```

Um dieses Problem zu umgehen, können Sie bei Bedarf schon beim Laden von Modellinstanzen angeben, dass zugehörige Elemente direkt mit geladen werden sollen – dieses Vorgehen nennt sich *Eager Loading*, »eifriges« Laden:

```
$contacts = Contact::with('phoneNumbers')->get();
```

Es reicht dazu, bei einer Abfrage die `with()`-Methode zu ergänzen; wie Sie in diesem Beispiel sehen können, übergibt man dabei einfach den Namen der Methode, durch die die Beziehung definiert ist.

Damit werden die vielen einzelnen Abfragen, die beim Lazy Loading entstehen (z.B. wenn die Telefonnummer eines Kontakts immer erst dann abgefragt wird, wenn sie in einer `foreach`-Schleife gebraucht wird), Laravel-intern auf nur noch zwei Abfragen reduziert: eine Abfrage, die die übergeordneten Elemente (hier die gewünschten Kontakte) liefert, und eine zweite, um alle verknüpften Elemente zu erhalten (alle Telefonnummern, die zu den gelieferten Kontakten gehören).

Sie können mehrere Beziehungen gleichzeitig »eifrig laden«, indem Sie mehrere Parameter an den Aufruf `with()` übergeben:

```
$contacts = Contact::with('phoneNumbers', 'addresses')->get();
```

Und Sie können Eager Loading auch verschachtelt anwenden, um Beziehungen von Beziehungen zu laden:

```
$authors = Author::with('posts.comments')->get();
```

Eager Loading einschränken. Wenn Sie zwar für eine Beziehung Eager Loading einsetzen, dabei aber nur ausgewählte Elemente berücksichtigen wollen, können Sie dies mit einer Closure erreichen, in der Sie eine entsprechende Bedingung festlegen:

```
$contacts = Contact::with(['addresses' => function ($query) {
    $query->where('mailable', true);
}])->get();
```

»Lazy« Eager Loading. Ich weiß, dass es ein bisschen verrückt klingt, weil wir Eager Loading gerade als eine Art Gegenteil von Lazy Loading definiert haben, aber manchmal kann es vorkommen, dass man nicht von vornherein weiß, ob man eine Abfrage mit Eager Loading durchführen will oder nicht. Indem man den Aufruf zum Laden der abhängigen Elemente vom Hauptaufruf trennt und erst später durchführt, kann man alle zugehörigen Elemente bei Bedarf nachladen und vermeidet dennoch die Nachteile des *N*+1-Problems. Das ist quasi »faules« Eager Loading:

```
$contacts = Contact::all();

if ($showPhoneNumbers) {
    $contacts->load('phoneNumbers');
}
```

Um Daten einer Beziehung nur dann zu laden, wenn dies nicht bereits geschehen ist, können Sie (seit Laravel 5.5) die Methode `loadMissing()` verwenden:

```
$contacts = Contact::all();

if ($showPhoneNumbers) {
    $contacts->loadMissing('phoneNumbers');
}
```

Eager Loading nur der Zeilenzahl

Wenn Sie nur die Gesamtzahl der abhängigen Elemente brauchen, können Sie dazu `withCount()` benutzen:

```
$authors = Author::withCount('posts')->get();

// Fügt jedem Autor einen Ganzzahl "posts_count" hinzu, in der die Anzahl
// der von diesem Autor geschriebenen Postings steht
```

Ereignisse in Eloquent

Eloquent-Modelle lösen jedes Mal, wenn bestimmte Aktionen stattfinden, sogenannte *Ereignisse* bzw. *Events* aus und senden eine entsprechende Nachricht in die Tiefen Ihrer Anwendung, unabhängig davon, ob die Anwendung gerade »zuhört« oder nicht. Falls Sie mit dem Pub-Sub-Muster vertraut sind: Es ist das gleiche Vorgehen (mehr über Laravels Event-System in Kapitel 16).

Wir geben einen kurzen Überblick, wie Sie einen Event Listener binden können, der immer dann reagieren soll, wenn ein neuer Kontakt erstellt wird. Wir werden den Listener in der `boot()`-Methode von `AppServiceProvider` verankern und nehmen im Beispiel an, dass wir jedes Mal einen Drittanbieterdienst benachrichtigen wollen, wenn ein neuer Kontakt erstellt wird.

Beispiel 5-62: Einen Listener an ein Eloquent-Ereignis binden

```
class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        $thirdPartyService = new SomeThirdPartyService;

        Contact::creating(function ($contact) use ($thirdPartyService) {
            try {
                $thirdPartyService->addContact($contact);
            } catch (Exception $e) {
                Log::error('Failed adding contact to ThirdPartyService; canceled.');

                return false; // Bricht die create()-Methode ab
            }
        });
    }
```

Was genau passiert in Beispiel 5-62? Zuerst verwenden wir *`Modelname`*`::`*`eventName()`* als Methode und übergeben ihr eine Closure. Diese Closure erhält Zugriff auf die Modellinstanz, auf der gearbeitet wird. Danach müssen wir diesen Listener in irgendeinem Service Provider definieren. Und drittens: Wenn wir `false` zurückgeben, müssen die Operation und der `save()`- bzw. `update()`-Vorgang abgebrochen werden.

Es folgen die Bezeichnungen der Ereignisse, über deren Eintritt ein Eloquent-Modell benachrichtigt:

- `creating`
- `created`
- `updating`
- `updated`
- `saving`
- `saved`
- `deleting`

- `deleted`
- `restoring`
- `restored`
- `retrieved`

Die meisten davon sollten relativ klar sein, bis vielleicht auf `restoring` und `restored`: Diese beiden Signale werden ausgelöst, wenn Sie eine Zeile wiederherstellen, die per Soft Delete gelöscht wurde. Außerdem wird `saving` sowohl bei `creating` als auch bei `updating` sowie `saved` bei `created` und `updated` ausgelöst.

`retrieved` (verfügbar ab Laravel 5.5) wird ausgelöst, wenn ein vorhandener Datensatz aus der Datenbank abgefragt wurde.

Testen

Laravel erleichtert durch sein Applikationstest-Framework das Testen Ihrer Datenbank – dazu müssen keine Unit-Tests für Eloquent geschrieben, sondern es kann direkt die gesamte Anwendung getestet werden.

Nehmen Sie folgendes Szenario: Sie möchten sicherstellen, dass eine bestimmte Seite ausschließlich einen ganz bestimmten Kontakt anzeigt. Die einem solchen Vorgang zugrunde liegende Logik hat mit dem Zusammenspiel zwischen der URL, dem Controller und der Datenbank zu tun, sodass ein Anwendungstest hier die beste Lösung ist. Vielleicht kommen Sie auch auf die Idee, Eloquent-Aufrufe zu mocken und zu vermeiden, dabei wirklich die Datenbank abzufragen. *Machen Sie das nicht.* Versuchen Sie es stattdessen mit Beispiel 5-63.

Beispiel 5-63: Testen von Datenbankinteraktionen mit einfachen Anwendungstests

```
public function test_active_page_shows_active_and_not_inactive_contacts()
{
    $activeContact = factory(Contact::class)->create();
    $inactiveContact = factory(Contact::class)->states('inactive')->create();

    $this->get('active-contacts')
        ->assertSee($activeContact->name)
        ->assertDontSee($inactiveContact->name);
}
```

Wie Sie sehen, eignen sich Modellfabriken und Laravels Anwendungstestfunktionen hervorragend zum Testen von Datenbankaufrufen.

Alternativ können Sie einen Datensatz auch direkt in der Datenbank suchen wie in Beispiel 5-64.

Beispiel 5-64: Mit assertDatabaseHas() nach bestimmten Datensätzen suchen

```
public function test_contact_creation_works()
{
    $this->post('contacts', [
```

```
        'email' => 'jim@bo.com'
    ]);

    $this->assertDatabaseHas('contacts', [
        'email' => 'jim@bo.com'
    ]);
}
```

Eloquent und Laravels Datenbank-Framework sind ausgiebig getestet worden. Sie müssen nicht von *Ihnen* getestet werden. Sie müssen sie nicht mocken. Wenn Sie wirklich vermeiden wollen, die Datenbank abzufragen, können Sie ein Repository-Entwurfsmuster verwenden und ungespeicherte Instanzen Ihrer Eloquent-Modelle zurückgeben. Entscheidend ist letztlich, dass Sie die Datenbanklogik Ihrer Anwendung testen.

Wenn Sie benutzerdefinierte Akzessoren, Mutatoren, Geltungsbereiche usw. einsetzen, können Sie auch diese ohne Umwege testen, wie Beispiel 5-65 zeigt.

Beispiel 5-65: Testen von Akzessoren, Mutatoren und Scopes

```
public function test_full_name_accessor_works()
{
    $contact = factory(Contact::class)->make([
        'first_name' => 'Alphonse',
        'last_name' => 'Cumberbund'
    ]);

    $this->assertEquals('Alphonse Cumberbund', $contact->fullName);
}

public function test_vip_scope_filters_out_non_vips()
{
    $vip = factory(Contact::class)->states('vip')->create();
    $nonVip = factory(Contact::class)->create();

    $vips = Contact::vips()->get();

    $this->assertTrue($vips->contains('id', $vip->id));
    $this->assertFalse($vips->contains('id', $nonVip->id));
}
```

Vermeiden Sie es einfach, Tests zu schreiben, bei denen Sie komplexe Methodenketten erstellen müssen, um zu behaupten, dass ein bestimmter Stapel auf einem Datenbank-Mock aufgerufen wurde. Wenn Ihre Tests des Datenbank-Layers zu erdrückend und komplex werden, liegt das möglicherweise daran, dass Sie sich von vorgefassten Vorstellungen in unnötig komplexe Strukturen zwingen lassen. Halten Sie den Ball einfach flach.

Unterschiedliche Bezeichnungen für Testmethoden vor Laravel 5.4

In Projekten mit Laravel-Versionen vor 5.4 sollte `assertDatabaseHas()` durch `seeInDatabase()`, `get()` durch `visit()`, `assertSee()` durch `see()` und `assertDontSee()` durch `dontSee()` ersetzt werden.

TL;DR

Laravel wird mit einer Reihe leistungsstarker Datenbank-Tools ausgeliefert, darunter Migrationen, Seeding, ein eleganter Query Builder und Eloquent, ein leistungsfähiges ActiveRecord-ORM. Auch ohne Eloquent kann man – mithilfe einer dünnen Komfortschicht – auf die Datenbank zugreifen und sie manipulieren, ohne selbst SQL-Statements schreiben zu müssen. Aber ein objektrelationaler Mapper wie Eloquent (oder Doctrine oder eine andere Alternative) lässt sich einfach hinzufügen und funktioniert problemlos mit den Datenbank-Tools, die Laravel von Haus aus mitbringt.

Eloquent folgt dem Active-Record-Muster, sodass es einfach ist, Klassen von datenbankgestützten Objekten zu definieren, einschließlich der Angabe der Tabellen, in denen sie gespeichert werden, und dem Aufbau der Spalten, Akzessoren und Mutatoren. Eloquent kann jede normale SQL-Anweisung durchführen und bewältigt auch komplexe Beziehungsmuster zwischen Modellen bzw. Tabellen bis hin zu polymorphen m:n-Beziehungen.

Laravel verfügt zudem über ein robustes System zum Testen von Datenbanken inklusive Modellfabriken.