Was mit Eloquent ORM möglich ist – erklärt am Beispiel Buchverleihtool

Laravel ORM im Einsatz

Eloquent, das ORM (Object-relational Mapping) von Laravel, erleichtert die Kommunikation mit einer Datenbank ungemein. Anhand eines JSON API für ein Buchverleihtool zeigen wir, was man damit anstellen kann – komplett unabhängig vom dahinterliegenden Datenbanksystem.

von Harald Nezbeda und Manuel Wutte

Laravel ist schon seit vielen Jahren eins der wohl beliebtesten PHP Frameworks. Das liegt u. a. daran, dass man mit Hilfe von Laravel schnell sowohl komplexe als auch sichere Webapplikationen erstellen kann. Der Entwicklungsprozess wird dadurch in vielen Bereichen – wie beispielsweise Routing, Caching, Authentifizierung, ORM u. v. a. m. – erheblich vereinfacht. Die zugehörige Dokumentation ist sehr gut gestaltet, klar strukturiert und bietet zahlreiche Codebeispiele, die selbst einem Laien den Einstieg verhältnismäßig einfach machen. Sie stellt den Grundpfeiler des Open-Source-Tools dar. Des Weiteren erfreut sich Laravel auch einer sehr großen und engagierten Community. In diesem Artikel werden wir uns auf Laravels Eloquent ORM konzentrieren und zeigen, wie man Eloquent in der Praxis einsetzt – komplett unabhängig vom dahinterliegenden Datenbanksystem.

Set-up mit Docker

Da wir auf unterschiedlichen Betriebssystemen unterwegs sind (Windows und Ubuntu Linux), haben wir uns dazu entschlossen, eine flexible Entwicklungsumgebung auf Basis von Docker zu nutzen. Die hat zusätzlich den Vorteil, dass sie uns den initialen Set-up-Prozess der einzelnen Komponenten, wie Datenbank und Webserver, deutlich vereinfacht. Die initiale Version des verwendeten Docker-Set-ups kann man auf GitHub [1] finden.

Datenbankdiagramm

Das Datenbankdiagramm (**Abb. 1**) haben wir mit Hilfe des Zeichentools draw.io erstellt. Wir exportieren dabei ein von draw.io generiertes SVG und speichern es ins Git

Repository des Projekts. Auf diese Weise können wir sicherstellen, dass wir immer ein Diagramm haben, das die aktuelle Codebasis widerspiegelt. Natürlich müssen wir selbst dafür Sorge tragen, dass wir jede Änderung des Datenbankschemas auch wieder korrekt im Diagramm abbilden.

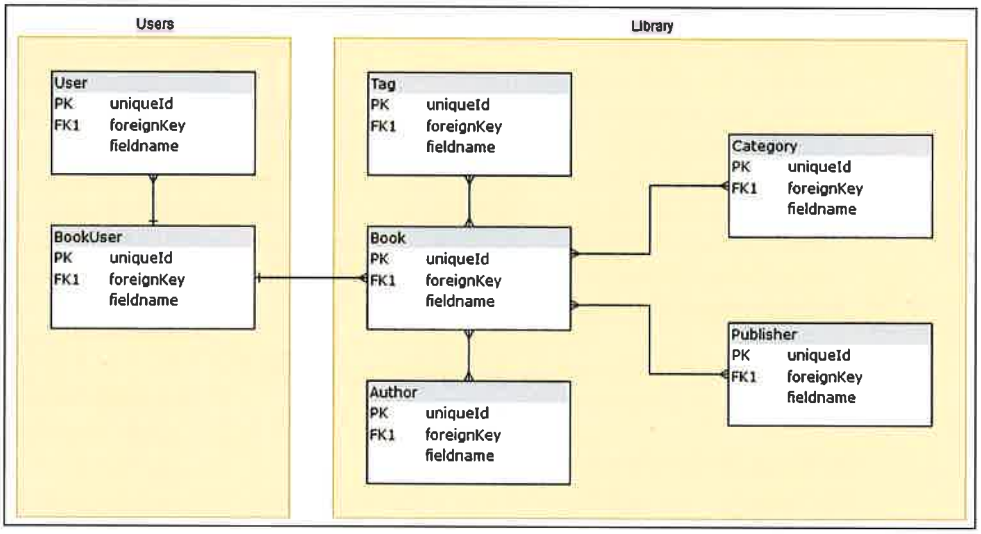

Abb. 1: Datenbankdiagramm [2]

Erstellen von Models und Migrationen

Migrationen dienen dazu, Kontrolle und Überblick über die gesamte Datenbankarchitektur zu bewahren. Zusätzlich sind damit schnelle und einfache Änderungen auf unterschiedlichsten Systemen möglich – ohne dass man sich Gedanken machen muss, ob man wohl eine Änderung übersehen hat. Mit dem Schema-Builder von Laravel wird das ganze Prinzip dahinter spielerisch einfach.

Um eine neue Migration zu erstellen, bietet sich das integrierte Kommandozeilenprogramm Artisan von Laravel an, mit dem sich Teile des Frameworks auch über das CLI steuern lassen. Dazu navigiert man mit der Konsole ins Root-Verzeichnis von Laravel und erstellt die neue Migration mit dem folgenden Command:

```
php artisan make:migration your_custom_migration_name
```

Möchte man sich eine Übersicht aller aktuell verfügbaren Artisan Commands mit zugehöriger Beschreibung anzeigen lassen, so geht das mit *php artisan*.

Hat man nun seine Migration erstellt, findet man im Verzeichnis *database/migrations* die Migrationsdatei. In ihr sind bereits zwei Methoden vorbereitet: *up()* und *down()*. Die *up*-Methode gibt an, was geschehen soll, wenn man die Migration ausführt, und die *down*-Methode, was im Fall eines Rollbacks unternommen werden soll. Wie eine solche Migration in der Praxis aussieht, zeigen wir am Beispiel einer Usertabelle des Laravel Authentication Skeleton (Listing 1).

Mit Hilfe von Artisan können nun mit *php artisan migrate* die Migrationen entsprechend chronologisch abgearbeitet und auf der Datenbank ausgeführt werden:

Laravel merkt sich nebenbei, welche der vorhandenen Migrationen bereits erfolgreich ausgeführt wurden, sodass es zu keiner doppelten Ausführung kommen kann. Ist beispielsweise bei der Entwicklung einer Migration ein Fehler in der Schemadefinition gemacht worden, aber die Migration wurde bereits ausgeführt, kann man mit dem folgenden Befehl ein Rollback der zuvor ausgeführten Migration vornehmen: *php artisan migrate:rollback*.

Aufbau des API

Wir haben uns dazu entschlossen, für das Beispielprojekt ein JSON API gemäß der Spezifikation (v1.0) von https://jsonapi.org zu bauen. Dafür haben wir uns das Package laravel-json-api [3] zunutze gemacht. Das Modul hilft uns, anhand eines fixen Sets aus Konfigurationen zu bestimmen, wie das API strukturiert werden soll. Jeder Endpunkt besteht aus den folgenden Dateien:

Listing 1

```
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' );
  }
}
```

- *Adapter.php* definiert die Verbindung zum Model und die Relationen zu anderen Models sowie Filtermöglichkeiten für den Listenendpunkt.
- *Schema.php* beinhaltet die Liste der Attribute des Models, die öffentlich sind und für die Serialisierung der Daten verwendet werden, sowie die Auflistung der Relationen, die mitgeladen werden können.
- *Validators.php* definiert die Liste der Relationen, die nachgeladen werden können, sowie die Regeln für die Validierung der Ressourcen.

Die vollständige Definition des API ist in unserem GitHub Repository [4] aufgeführt.

Befüllung des Systems mit Testdaten

Um Testdaten auf eine schnelle und einfache Art und Weise generieren zu können, haben wir uns für die PHP Library Faker entschieden. Sie ist genau auf solche Zwecke ausgelegt und wird auch häufig für die Datenbefüllung im Zuge von Stresstests oder zur Anonymisierung von Daten aus Produktivsystemen herangezogen. Damit Faker weiß, welche Felder es mit welchen Daten befüllen soll, muss man das zuerst mit sogenannten Model Factories definieren (Listing 2).

Wie man anhand dieses Beispiels schön sieht, wird als Index das Feld aus der Datenbank definiert und als Wert die Art von Daten, die Faker hierfür verwenden soll. Zu erkennen ist auch, welche Vielfalt von Mock-Daten von Faker generell bereitgestellt wird. Damit eine Befüllung mit Testdaten jederzeit möglich ist, haben wir uns entschlossen, die folgende Faker-Logik in einem eigenen Artisan Command zu verpacken (Listing 3). Genauere Details dazu findet ihr in unserem GitHub Repository [4].

ORM-Operationen

Das JSON API beherrscht nun die wesentlichen CRUD-Operationen – also das Anlegen (Create), Lesen (Read), Aktualisieren (Update) und Löschen (Delete) von Datensätzen. Um eine vollwertige Frontend-Applikation (z. B. mit Angular, Vue oder React) zu bauen, müssen wir die Endpunkte noch entsprechend erweitern. Ebenso fehlen noch Filtermöglichkeiten oder Sortieroptionen. Auch oft benötigt werden zusätzliche Endpunkte, an denen Daten aggregiert zurückgeliefert werden.

Das Erstellen dieser fehlenden Endpunkte und Filter dient uns als typisches Beispiel, an dem wir die Stärken und Möglichkeiten des Eloquent ORM demonstrieren können. Die Filter werden in der *filter*-Methode des Adapters für das Book Model implementiert, wo wir auf die Query zugreifen können (Listing 4).

Filtern nach Namen

Die *where*-Methode macht das Filtern nach Eigenschaften, die sich direkt am Objekt befinden, einfach. Damit wir auch bei einer Teilübereinstimmung ein Ergebnis erhalten, wird die *like*-Operation verwendet. Zusätzlich dazu erweitern wir den Suchbegriff noch mit der SQL Wildcard %, wie hier zu sehen ist:

```
$query->where('name', 'like', "%{$name}%");
```

Listing 2

```
use Faker\Generator as Faker;

$factory->define( \App\Models\Book::class, function ( Faker $faker ) {
  return [
    'name'           => $faker->sentence,
    'pages'          => $faker->numberBetween( 50, 600 ),
    'ean'            => $faker->ean13,
    'isbn_10'        => $faker->isbn10,
    'isbn_13'        => $faker->isbn13,
    'release_date'   => $faker->dateTime,
    'type'           => $faker->randomElement( [ 'HARDCOVER', 'SOFTCOVER', 'EBOOK' ] ),
    'nsfw'           => $faker->boolean,
    'publisher_id'   => \App\Models\Publisher::inRandomOrder()->first()->id,
  ];
} );
```

Listing 3

```
factory( Tag::class, 50 )->create();
factory( Category::class, 15 )->create();
factory( Publisher::class, 20 )->create();
factory( Author::class, 80 )->create();

factory( Book::class, 200 )->create()->each( function ( $a ) {
  $a->authors()->attach( Author::all()->random( rand( 1, 5 ) ) );
  $a->categories()->attach( Category::all()->random( rand( 1, 3 ) ) );
  $a->tags()->attach( Tag::all()->random( rand( 0, 10 ) ) );
} );

factory( User::class, 50 )->create();

$users = User::inRandomOrder()->limit( 10 )->get();
foreach ( $users as $user ) {
  $books = Book::inRandomOrder()->limit( 10 )->get();

  foreach ( $books as $book ) {
    $bookUser          = new BookUser();
    $bookUser->book_id = $book->id;
    $bookUser->user_id = $user->id;
    $bookUser->save();

  }
```

Listing 4

```
class Adapter extends AbstractAdapter {
  public function __construct( StandardStrategy $paging ) {
    parent::__construct( new \App\Models\Book(), $paging );
  }

  protected function filter( $query, Collection $filters ) {
  }
}
```

Filtern nach Publisher-ID

In unserem Beispiel des Buchverleihtools besteht zwischen Buch und Publisher eine *1:n*-Verbindung. Somit ist die ID des Publishers direkt auf dem Buch verfügbar. Das bedeutet, man kann auch nach dem Feld *publisher_id* filtern:

```
$query->where('publisher_id', '=', $publisher);
```

Filtern nach Kategorie/Tags/Autor oder User-ID

Bei den Kategorien Tags und Autoren haben wir es jedoch mit *m:n*-Verbindungen zu tun. Um das abbilden zu können, müssen wir eine Ebene tiefer gehen. Dazu machen wir uns das *whereHas*-Query zunutze. Somit können wir direkt auf dem verbundenen Objekt arbeiten:

```
$query->whereHas('categories', function($q) use($category) {
  $q->where('id', '=', $category);
});
```

Filtern nach Publisher-Kurzname

In unserer Datenbank besitzt der Publisher zusätzlich einen Kurznamen, nach dem wir ebenfalls filtern möchten. Der Filter soll immer exakt nach dem Wort suchen, aber Groß- und Kleinschreibung ignorieren. Dazu können wir ebenfalls die *whereHas*-Methode verwenden und so direkt auf das verbundene Objekt zugreifen. Das wird dann mittels der SQL-Methode *LOWER* und *strtolower* von PHP in ein *whereRaw* verpackt:

```
$query->whereHas('publisher', function($q) use($publisherShortName) {
  $q->whereRaw('LOWER(short_name) = ? ', [trim(strtolower($publisherShort
                                                    Name))]);
});
```

Filtern nach Autorenname

Um im Beispieltool nach dem Namen eines Autors filtern zu können, verbinden wir den vorherigen Filter mit jenem zur Filterung nach dem Namen des Buchs:

```
$query->whereHas('authors', function($q) use($authorName) {
  $q->whereRaw('LOWER(name) LIKE ? ', ['%'.trim(strtolower($authorName)). '%']);
});
```

Filtern nach Verfügbarkeit

Wir gehen davon aus, dass jedes entliehene Buch einem User zugeordnet ist. Ein Buch ist immer dann verfügbar, wenn es keinem User zugeordnet ist. So filtern wir dementsprechend nach Büchern, die verfügbar sind: *$query->doesntHave('users');*

Und so filtern wir nach Büchern mit einem verknüpften User, die entsprechend ausgeliehen bzw. nicht verfügbar sind: *$query->whereHas('users');*

Im API haben wir hierfür einen Filter implementiert, der die als Strings übergebenen Werte in boolesche Werte konvertiert. Das ist notwendig, weil die Queryparameter als Strings übergeben werden und wir richtige Booleans benötigen.

Filtern nach Veröffentlichungsdatum (Jahr/Monat/Tag)

Jedes Buch besitzt ein Veröffentlichungsdatum, und selbstverständlich wollen wir auch das zum Filtern heranziehen. Uns interessiert dabei nicht ein bestimmtes volles Datum, sondern meistens nur das Jahr, in dem das Buch veröffentlicht wurde – oder maximal zusätzlich der Monat. Eloquent bietet für solche Datumsabfragen bereits fertige Methoden:

```
$query->whereYear('release_date', $releaseYear);
$query->whereMonth('release_date', $releaseMonth);
$query->whereDay('release_date', $releaseDay);
```

Zufälliges Buch (gefiltert nach Kategorie/Tags/Autor)

Sehr praktisch an Eloquent ist auch, dass man dem Benutzer die Möglichkeit geben kann, ein zufälliges Buch auszuwählen. Das ORM von Laravel bietet auch für diese Situation bereits eine fertige Methode an, die uns bei der zufälligen Sortierung zugutekommt: *$query->inRandomOrder();*

Wir können diese Random-Sortierungsmethode in weiterer Folge auch mit den bisherigen Filtern kombinieren, um so z. B. ein beliebiges Buch aus dem Jahr 2007 zu finden, das aktuell noch verfügbar ist:

```
$query->inRandomOrder()->doesntHave('users')->whereYear('release_date',
                                                    2007)->first();
```

Aggregation: Statistik der Bücher

Ein anderes Thema, das von Eloquent ebenfalls sehr gut umgesetzt wurde, ist die Aggregation von Daten. Wir haben dafür einen eigenen Endpunkt gebaut, da wir hier vom Ressourcenschema des JSON API abweichen. Mit Hilfe der *count*-Methode kann man leicht die Anzahl der Einträge pro Model herausfinden:

```
'books_count' => Book::count(),
'books_available_count' =>
  Book::doesntHave( 'users' )->count(),
'books_borrowed_count' =>
    Book::whereHas( 'users' )->count(),
```

Das gleiche Prinzip lässt sich auch auf verknüpfte Objekte anwenden:

Listing 5

```
$books = Book::all()
foreach ($books as $book) {
  $book->publisher->name;
  $book->authors->count();
  $book->users->count();
  $book->tags->count();
  $book->categories->count();
}
```

Listing 6

```
namespace App\Models;

use Illuminate\Database\Eloquent\Builder;

class Book extends BaseModel {
  protected static function boot() {
    parent::boot();

    if (static::workTime()) {
      static::addGlobalScope('nsfw',
            function (Builder $builder) {
        $builder->where('nsfw', '=', false);
      });
    }
  }
}
```

```
'tags' => Tag::select( 'id', 'name' )->withCount( [ 'books', 'books_available',
                'books_borrowed' ] )->orderBy( 'books_count', 'DESC' ) ->get(),
```

Bei der Anwendung dieser Methode ist zu beachten, dass *select* unbedingt vor *count* stehen muss, um so im Tupel nur die ID und den Namen zurückzubekommen. Das nachfolgende *withCount* sorgt dafür, dass wir die entsprechende Aggregation auch auf der Collection haben. Die Aggregation kann auch mit weiteren Bedingungen kombiniert werden. In unserem Fall wollen wir beispielsweise weitere User kombiniert mit der Anzahl der von ihnen entliehenen Bücher finden, aber dabei nur diejenigen, die tatsächlich Bücher ausgeliehen haben:

```
User::select( 'id', 'name' )->withCount( 'books' )->has( 'books', '>',
                0 )->orderBy( 'books_count', 'DESC' )->get(),
```

Details zum Buch (Includes) – Eager Loading

Dieser Teil wird bereits vom Package, das wir für die Umsetzung des API verwendet haben, übernommen. Es ist dennoch wichtig zu wissen und zu verstehen, was Eloquent hier genau macht, und auch warum – sollte man ein solches Verhalten mal selbst implementieren müssen.

Der folgende Codeblock in Listing 5 verursacht beispielsweise insgesamt 1001 Datenbankabfragen (1 Query für Books, 200 Querys jeweils für Publisher, Authors, Users, Tags und Categories, da wir 200 Bücher in der Datenbank haben):

Mittels Eager Loading kann diese Datenbankabfrage erheblich optimiert werden. Ein Beispiel mit Eager Loading:

```
$books = Book::with(['publisher', 'authors', 'users', 'tags', 'categories'])->get();
```

Ein zweites Beispiel mit Lazy Eager Loading:

```
$books = Book::all()->load(['publisher', 'authors', 'users', 'tags', 'categories']);
```

Wenn man diesen Codeschnipsel nun debuggt, wird man sehen, dass die Anzahl der Datenbankabfragen nun auf 6 (!) von ursprünglich 1001 reduziert werden konnte. Das hat natürlich enorme Auswirkungen auf die Geschwindigkeit und verbessert die Performance der Applikation deutlich.

Ausblenden von Büchern, die NSFW sind

Wir haben die Implementierung soweit finalisiert. In unserem Beispiel wünscht der Auftraggeber jedoch zusätzlich, dass Bücher, die aufgrund ihres Inhalts nicht zu Arbeitszeiten angezeigt werden sollten, ausgeblendet werden. Meistens sind solche nachträglichen Implementierungen mit viel Refactoring-Arbeit und daher mit hohem zeitlichen Aufwand verbunden. Doch mit Hilfe sogenannter Global Scopes kann diese Funktionalität (zumindest hier in unserem Fall) sehr schnell und einfach hinzugefügt werden (Listing 6).

Fazit

Mit Eloquent besitzt das Laravel Framework ein überaus mächtiges Datenbankabstraktionstool, das es Entwicklern sehr einfach macht, schnell und flexibel mit unterschiedlichsten relationalen Datenbanken (MySQL, PostgreSQL, SQLite und SQL Server) zu interagieren. Durch seinen hohen Abstraktionsgrad ist selbst ein Wechsel des DBMS im Produktivbetrieb meist ohne größere Probleme möglich – solange man auf die von Eloquent bereitgestellten Methoden zurückgreift und Raw-Querys nach Möglichkeit weitestgehend vermeidet. Für jede Datenbanktabelle wird bei diesem ORM-Konzept ein eigenes entsprechendes Model definiert, über das die Kommunikation zwischen Applikation und Datenbank abläuft. Wenn man sich dabei an ein paar wenige, leicht zu merkende Konventionen bei der Namensgebung der einzelnen Datenbanktabellen und Models hält, kann nichts schiefgehen. Mit einer riesigen Auswahl unterschiedlicher vorgefertigter Methoden fällt auch das Erstellen von komplexen Abfragen verhältnismäßig einfach.

Da das Eloquent ORM selbst auf einer ActiveRecord-Implementierung basiert, müssen Models und Migrationen jeweils selbst definiert und geschrieben werden. Gleiches gilt auch für das Definieren der einzelnen Beziehungen (Relations) der Records untereinander. Mit Hilfe des enthaltenen Kommandozeilenprogramms Artisan kann aber auch hier ein Großteil der notwendigen Arbeiten, wie z. B. das Anlegen von Migrationen oder auch das Erstellen von Model Skeletons, erheblich vereinfacht werden. Gleiches gilt auch für die Interaktion mit der Datenbank in Bezug auf das Migrationsverhalten und den damit verbundenen Möglichkeiten.

Um euch einen tieferen Einblick in diese Materie zu gewähren, und auch um die praktische Integration leichter zu verstehen, haben wir den gesamten Quellcode, den wir für dieses Beispielprojekt geschrieben haben, auf GitHub [4] veröffentlicht und euch unter der MIT-Lizenz zur Verfügung gestellt.

Harald Nezbeda ist 2013 als Software Developer bei Anexia tätig. Erfahrung in der Entwicklung von Webapplikationen auf Basis von Zend, Laravel, CodeIgniter, Apache Struts und AngularJS unter Nutzung von PHP, Java und JavaScript.

Manuel Wutte ist Senior Software Developer und Technical Leader bei Anexia. Über zehn Jahre Erfahrung in der Entwicklung von Software mit unterschiedlichen Webtechnologien und Frameworks wie z. B. Laravel, Django und Angular.

Links & Literatur

[1] https://github.com/nezhar/laravel-docker-compose

[2] https://github.com/anx-hnezbeda/laravel-book-corner/blob/master/docs/schema.svg

[3] https://laravel-json-api.readthedocs.io/en/latest/

[4] https://github.com/anx-hnezbeda/laravel-book-corner