Jan Teriete

Objektorientiertes PHP7

Band 2: MySQL und Doctrine 2

Ein Webmasters Press Lernbuch

Version 10.4.0 vom 23.04.2019

Autorisiertes Curriculum für das Webmasters Europe Ausbildungs- und Zertifizierungsprogramm

— www.webmasters-europe.org —

Inhaltsverzeichnis

6 PHP-Objekte mit Doctrine speichern

In dieser Lektion lernen Sie

- wie das SchemaTool Ihnen bei der Installation Ihrer Anwendung hilft.
- wie Sie mit Doctrine Datensätze in die Datenbank einfügen.
- wieso Doctrine beim Speichern mehrerer Datensätze sehr performant ist.

6.1 Einleitung

Als nächstes wenden wir uns dem Speichern von Objekten zu. Genaugenommen besteht dieser Vorgang aus zwei unterschiedlichen Operationen, da in SQL das Einfügen eines neuen Datensatzes (`INSERT`) ein anderes Statement ist als das Aktualisieren eines vorhandenen Datensatzes (`UPDATE`).

In dieser Lektion werden wir zunächst einmal nur das `INSERT` am Beispiel der `Tag`-Entity betrachten, doch ich kann jetzt schon verraten, dass bei Doctrine kein wesentlicher Unterschied zwischen beiden Operationen besteht.

6.2 Das SchemaTool

Zunächst benötigen wir für diese Speicherung noch eine Tabelle in unserer Datenbank. Doctrine stellt uns zu diesem Zweck als wirklich nettes Goodie die Klasse `Doctrine\ORM\Tools\SchemaTool` zur Verfügung, die anhand unserer Annotationen alle benötigten Tabellen anlegen kann.

Beispiel

```php
<?php

require_once 'inc/bootstrap.inc.php';

$schemaTool = new \Doctrine\ORM\Tools\SchemaTool($em);

$factory = $em->getMetadataFactory();
$metadata = $factory->getAllMetadata();

try {
    $schemaTool->updateSchema($metadata);
} catch (PDOException $e) {
    echo 'ACHTUNG: Bei der Aktualisierung des Schemas gab es ein Problem: ';
    echo $e->getMessage() . "<br />";
    if (preg_match("/Unknown database '(.*)'/", $e->getMessage(), $matches)) {
        die(
            sprintf(
                'Erstellen Sie die Datenbank %s mit der Kollation
utf8_general_ci!',
                $matches[1]
            )
        );
    }
```

```
23 }
24
25 ?>
26 Das Schema-Tool wurde durchlaufen.
```

Codebeispiel 26 *setup.php*

Die *setup.php* ist eine Art Mini-Controller, der lediglich für eine einzige sehr spezielle Aktion zuständig ist und keine Templates benötigt. Ab Zeile 10 finden Sie in diesem Controller Code, der eine erfolgreiche Abarbeitung von Zeile 11 überprüft. Hierfür wird die in PHP 5 eingeführte Ausnahmebehandlung[42] genutzt. Die wirklich relevanten Code-Zeilen sind somit nur 3 bis 8 und 11. Die restlichen Zeilen dienen lediglich der Fehlerkontrolle und -ausgabe. In Zeile 3 aktivieren wir wie im Front-Controller *index.php* das Bootstrapping. Danach instanziieren wir in Zeile 5 das ***SchemaTool***. Als Parameter benötigt dessen Konstruktor den ***EntityManager*** `$em`. Anschließend rufen wir in Zeile 7 die Methode `EntityManager#getMetadataFactory()` auf. Diese Methode hat als Rückgabewert eine Instanz der Klasse `Doctrine\ORM\Mapping\ClassMetadataFactory`. Von diesem Objekt rufen wir wiederum in Zeile 8 die Methode `ClassMetadataFactory#getAllMetadata()` auf. Hiermit erhalten wir die sogenannten Metadaten. Dies sind primär die Informationen über unsere Datenbank-Struktur, welche wir in Form von Annotationen festgelegt haben. Diese Metadaten benutzen wir dann in Zeile 11, um das Datenbank-Schema zu aktualisieren und so die benötigte Tabelle anzulegen.

Übung 9:

1. Implementieren Sie den in dieser Lektion vorgestellten Mini-Controller *setup.php*.
2. Rufen Sie den Mini-Controller im Browser auf. Achten Sie darauf, dass nach dem Ausführen die Tabelle `tags` tatsächlich in der Datenbank existieren muss.

Sollten Sie die Meldung `PDOException: SQLSTATE[42000] [1049] Unknown database` erhalten, so existiert die unter `$connectionOptions` in der *config.inc.php* angegebene Datenbank (noch) nicht. Oftmals liegt dies beispielsweise an einem Buchstabendreher.

6.3 Das Controller-Skeleton

Wir benötigen nun einen vollwertigen Controller, von dem aus wir die eigentlichen Aktionen ausführen können. Unsere Projektstruktur enthält deswegen bereits eine noch ziemlich leere Controller-Klasse `IndexController`.

Beispiel

```php
1  <?php
2
3  namespace Controllers;
4
5  class IndexController extends AbstractBase
6  {
7      public function indexAction()
8      {
9          // Fill me
10     }
11 }
```

Codebeispiel 27 *src/Controllers/IndexController.php (Version 1 - Controller-Skeleton)*

42. http://php.net/de/language.exceptions

Das hier vorgestellte **Controller-Skeleton** (dt.: Skelett bzw. Vorlage) sollte für Sie keine Überraschungen mehr bereithalten. Wichtig ist lediglich, dass in Zeile 3 der korrekte Namespace `Controllers` verwendet wird. Diesen haben wir schließlich in der *composer.json* bei der Konfiguration des Autoloadings angegeben. Außerdem nutzen wir wie schon in »Band 1: Grundlagen der OOP« eine Vererbung von der Elternklasse `AbstractBase`. Diese Elternklasse enthält allerdings ein paar Neuerungen, auf die ich erst nach und nach zu sprechen komme.

6.4 Das eigentliche Speichern

Nun legen wir eine Instanz der Klasse `Tag` in der Methode `indexAction` an und befüllen die Attribute mit sinnvollen Werten. Da die ID automatisch befüllt werden soll und es deswegen für das Attribut `$id` keinen Setter gibt, müssen wir nur das Attribut `$title` befüllen.

Beispiel

```
1  <?php
2
3  namespace Controllers;
4
5  use Entities\Tag;
6
7  class IndexController extends AbstractBase
8  {
9      public function indexAction()
10     {
11         $tag = new Tag(
12             ['title' => 'PHP']
13         );
14
15         $this->addContext('tag', $tag);
16     }
17 }
```

Codebeispiel 28 *src/Controllers/IndexController.php (Version 2)*

Das Objekt wird mit der Methode `addContext` in dem Attribut `$context` abgelegt, wodurch wir im Template *indexAction.tpl.php* unter der Variable `$tag` Zugriff auf das Objekt haben.

Beispiel

```
1 <p>
2     Es wurde das Tag <strong><?= $tag ?></strong>
3     mit der ID <?= $tag->getId() ?> angelegt.
4 </p>
```

Codebeispiel 29 *templates/IndexController/indexAction.tpl.php*

Wenn Sie nach diesen Änderungen unseren Front-Controller im Browser aufrufen, so erhalten Sie (noch) die Ausgabe, dass ein Datensatz mit der ID `0` angelegt wurde.

Wir verwenden für Templates grundsätzlich die doppelte Dateiendung *.tpl.php* und zur Ausgabe in diesen Template-Dateien die sogenannte **echo-Shortcut-Syntax** mit `<?=` (anstatt `<?php echo`).

Wir verzichten derzeit (noch) auf den Einsatz von `htmlspecialchars()` oder `strip_tags()` bei der Ausgabe, um die Beispiele kurz zu halten. Dies ändert sich erst wieder im nächsten OOP-Lernbuch »Band 3: Eine Einführung in das Thema Sicherheit«, wenn wir uns intensiver mit der Sicherheit von Anwendungen auseinandersetzen.

6.4.1 persist

Als nächsten Schritt übergeben wir das eben erzeugte Objekt dem `EntityManager` zum Speichern. Dies geschieht über die Methode `EntityManager#persist()`, der Sie das Objekt als Parameter übergeben.

Beispiel

```
$em = $this->getEntityManager();
$em->persist($tag);
```

Codebeispiel 30 *IndexController.php - Erweiterung der Standardaktion*

6.4.2 flush oder das Entwurfsmuster Unit of Work

Anders als Sie nun vielleicht erwarten, führt Doctrine das SQL-`INSERT` nicht an der Stelle aus, an der Sie die Methode `EntityManager#persist()` aufrufen. Doctrine sammelt Anweisungen standardmäßig und führt sie am Stück aus. Auf diese Weise können Sie in einem Rutsch z. B. gleich mehrere Datensätze anlegen, was wesentlich performanter ist als mit einzelnen SQL-Anweisungen.

Erst wenn Sie die Methode `EntityManager#flush()` aufrufen, werden alle SQL-Anweisungen ausgeführt, die sich bis zu diesem Zeitpunkt angesammelt haben, egal ob es nur eine oder gleich hundert sind. Dieses Vorgehen, Aufgaben zu sammeln und am Stück abzuarbeiten, nennt man ***Unit of Work***. Es ist ein bekanntes Entwurfsmuster nicht nur im Datenbank-Bereich.

Beispiel

```
1  <?php
2
3  namespace Controllers;
4
5  use Entities\Tag;
6
7  class IndexController extends AbstractBase
8  {
9      public function indexAction()
10     {
11         $tag = new Tag(
12             ['title' => 'PHP' . time()]
13         );
14
15         $em = $this->getEntityManager();
16         $em->persist($tag);
17         $em->flush();
18
19         $this->addContext('tag', $tag);
20     }
21 }
```

Codebeispiel 31 *src/Controllers/IndexController.php (Version 3)*

Wenn Sie den Front-Controller *index.php* mehrfach aufrufen, werden Sie feststellen, dass das Attribut `$id` automatisch hochgezählt wird. Nach dem Speichern fügt Doctrine nämlich automatisch die korrekte ID in das Objekt ein.

Sollten Sie beim mehrfachen Aufruf die Meldung `SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate entry` erhalten, so haben Sie für den Alternativschlüssel `title` einen bereits in der Datenbank vorhandenen Wert verwendet.

Ihnen ist also eine kleine Änderung in Zeile 12 der Controller-Klasse entgangen. Falls Sie diese Meldung trotz identischem Wert nicht erhalten, so haben Sie die `unique`-Annotation beim entsprechenden Attribut der `Tag`-Klasse vergessen.

6.5 Zusammenfassung

Gratuliere, dies waren Ihre ersten sinnvollen Doctrine-Anweisungen. Ist Ihnen aufgefallen, dass Sie keine einzige Zeile SQL geschrieben und dennoch ein `INSERT`-Statement ausgeführt haben?

6.6 Testen Sie Ihr Wissen!

1. Reicht es, die Methode `EntityManager#persist()` aufzurufen, um ein Objekt zu speichern?
2. Wie nennt man das Vorgehen, Aufgaben zu sammeln und am Stück abzuarbeiten?

6.7 Übungen

Übung 10:

Implementieren Sie in Ihrem Projekt die in dieser Lektion vorgestellte *IndexController.php* und die dazu passende *indexAction.tpl.php*.

Übung 11:

Testen Sie die Controller-Klasse und rufen Sie dazu den Front-Controller *index.php* im Browser auf.

6.8 Optionale Übungen

Übung 12:

Erstellen Sie einen neuen Mini-Controller *reset.php*. Dieser soll drei verschiedene Tags (z. B. HTML, JavaScript und PHP) als Datensätze in der Datenbank ablegen. Dieser Controller ist vergleichbar mit der *setup.php*. Er ist nur für eine einzige Aktion zuständig und verwendet keine Templates. Geben Sie am Schluss der Datei lediglich eine Meldung aus, damit Sie beim Aufruf nicht eine komplett leere Seite angezeigt bekommen.

Übung 13:

Testen Sie den Controller *reset.php* durch einen Aufruf im Browser. Was passiert? Rufen Sie den Controller erneut auf. Was passiert nun?

Übung 14:

Spätestens beim zweiten Aufruf des Controllers wird versucht, ein schon vorhandenes `Tag` erneut anzulegen. Dies wird mit der schon bekannten Fehlermeldung quittiert. Da der Controller jedoch tatsächlich einen kompletten Reset unserer Datenbankinhalte ermöglichen soll, ergänzen wir vor dem Persist ein Truncate mit dem Befehl `$em->getConnection()->query('TRUNCATE TABLE tags;')`.