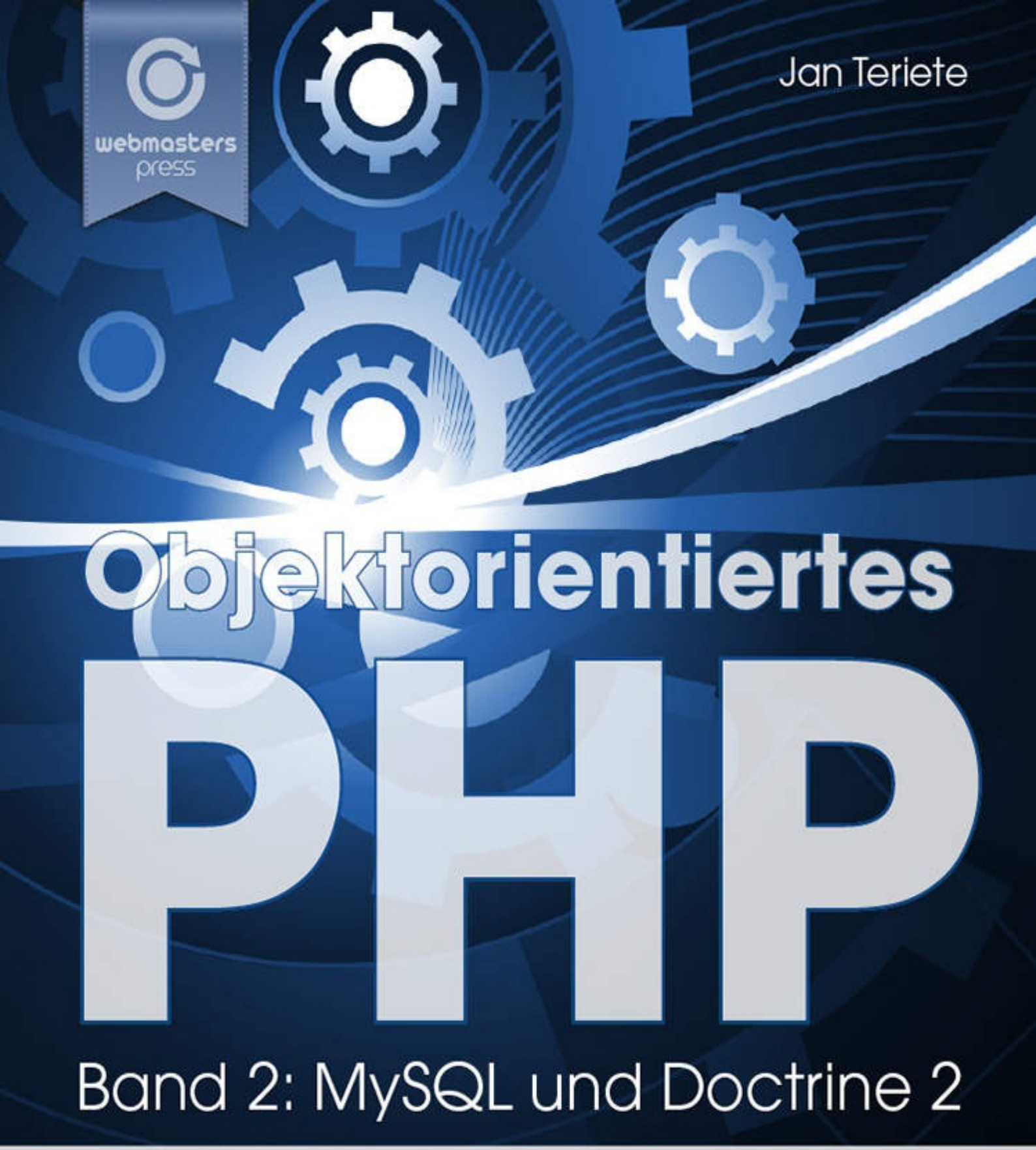

- Erste Schritte mit Composer und Packagist
- Praktischer Einstieg in Doctrine 2
- Sicherheit von PHP-Anwendungen
- Mit vielen Übungen und Code-Beispielen

Autorisiertes Curriculum für das **Webmasters Europe** Ausbildungs- und Zertifizierungsprogramm

we

Jan Teriete

Objektorientiertes PHP

Band 2: MySQL und Doctrine 2

Ein Webmasters Press Lernbuch

Version 8.0.0 vom 18.1.2017

Autorisiertes Curriculum für das Webmasters Europe Ausbildungs- und Zertifizierungsprogramm.

www.webmasters-europe.org

www.webmasters-press.de

Webmasters Akademie Nürnberg GmbH
Neumeyerstr. 22–26
90411 Nürnberg
Germany
www.webmasters-akademie.de

Printed books made with Prince

Art.-Nr. 12e80c5577b2
Version 8.0.0 vom 18.1.2017

Inhaltsverzeichnis

1 Einführung

In dieser Lektion lernen Sie
• welche Ziele dieses Lernbuch hat. • was Sie an Wissen mitbringen sollten, um erfolgreich mit diesem Lernbuch arbeiten zu können.

1.1Einleitung

Dieses Buch versucht, Ihnen eine neue Sichtweise auf die PHP-Programmierung zu vermitteln. Bisher haben Sie wahrscheinlich vorwiegend kleinere PHP-Projekte mittels prozeduraler Programmierung umgesetzt. Je größer ein Projekt allerdings wird, desto wichtiger, aber auch schwieriger ist es, die Übersicht zu behalten. Ihre Fähigkeiten in diesem Bereich zu verbessern, ist das Ziel dieses Lernbuchs.

Natürlich werden Sie auch neue PHP-Techniken erlernen. Sie lernen verbreitete ORM-Entwurfsmuster (objekt-relationales Mapping) kennen und werden in ein paar Code-Bibliotheken rund um das Doctrine-Projekt eingeführt. Weiter werden Sie sich bei der Verwaltung von Drittanbieter-Abhängigkeiten mit Composer anfreunden dürfen, und Sie erfahren natürlich auch, wie Sie mit Doctrine objektorientiert auf MySQL-Datenbanken zugreifen können.

Im wichtigeren Teil dieses Buches geht es jedoch darum, wie Sie OOP einsetzen können, um sauberen, wartbaren und vor allem lesbaren Code zu schreiben, und welche Sicherheitsprobleme Sie auf jeden Fall bei der Umsetzung beachten sollten.

1.2 Vorkenntnisse

Für die Arbeit mit diesem Lernbuch sollten Sie folgende Fähigkeiten mitbringen:

- Sie sollten die grundlegenden PHP-Techniken beherrschen (Variablen, Konstanten, Zuweisungen, Blöcke usw.).
- Sie sollten mit den Datentypen Integer, Float, String und Array umgehen können.
- Sie sollten die gängigen Kontrollstrukturen wie Schleifen (`for`, `while`, `foreach`) und Verzweigungen (`if-else`, `switch`) beherrschen.
- Sie sollten sich mit den Grundlagen der Objektorientierung beschäftigt, also beispielsweise den ersten Band dieses Lernbuchs durchgearbeitet haben.
- Sie sollten ***Model-View-Controller*** kennen und insbesondere eine Vorstellung davon haben, wie sich MVC-Konzepte mit PHP umsetzen lassen.
- Sie sollten Funktionen und Methoden schreiben können, die Parameter erwarten und Rückgabewerte erzeugen.
- Sie sollten gängige PHP-Funktionen beherrschen und sich bei Bedarf selbstständig mittels http://php.net neue aneignen können.
- Sie sollten möglichst etwas Erfahrung in der Entwicklung kleiner oder mittlerer Webapplikationen mitbringen.
- Sie sollten wissen, was Programmierrichtlinien sind und diese auch konsequent einhalten können.
- Sie sollten auch etwas Erfahrung im Umgang mit Datenbanken mitbringen, vorzugsweise MySQL, da sich die Beispiele im Buch auf MySQL beziehen.
- Sie sollten auf jeden Fall wissen, wie Sie eine MySQL-Datenbank auf Ihrem Entwicklungsrechner anlegen.
- Sie sollten ein Konsolen-Fenster öffnen können und wissen, wie Sie dort in ein bestimmtes Verzeichnis wechseln.
- Sofern Sie Linux oder Mac OS X als Entwicklungsplattform verwenden, sollten Sie Datei- bzw. Verzeichnisberechtigungen anpassen können.

Bei Windows ist mit dem Konsolen-Fenster die Eingabeaufforderung gemeint und bei Mac OS X die Terminal.app aus den Dienstprogrammen. Unter Windows lässt sich der Verzeichniswechsel in der Eingabeaufforderung übrigens ohne jede Tipparbeit erledigen, indem Sie

zuerst den Windows-Explorer starten und per Mausklick den gewünschten Ordner öffnen. Halten Sie jetzt die SHIFT-Taste gedrückt und klicken Sie mit der rechten Maustaste auf den Ordner. Im Kontextmenü erscheint der zusätzliche Befehl **Eingabeaufforderung hier öffnen**. Damit öffnet Windows ein Konsolen-Fenster und springt sofort in den angeklickten Ordner.

Um das Wichtigste nicht zu vergessen: Sie sollten natürlich auch Spaß daran haben, an Ihrem Code herumzuknobeln und neue Dinge auszuprobieren.

1.3 Aufbau der Lektionen

Zu Beginn jeder Lektion gibt es einen Abschnitt, der Ihnen die Lernziele vermittelt. Meistens werden Sie vor ein konkretes Problem gestellt oder es wird Ihnen zumindest gezeigt, an welchen Stellen Sie mit Ihren momentanen Kenntnissen Schwierigkeiten haben. Im Rest der Lektion erarbeiten Sie sich dann Schritt für Schritt die für die Lösung notwendigen Kenntnisse und Fähigkeiten.

1.3.1 Aufgaben im Fließtext

Sie werden immer wieder mitten im Text auf Aufgaben stoßen. Diese erfordern normalerweise nur einen recht geringen Zeitaufwand. Sie dienen entweder als Vorbereitung auf kommende Inhalte oder sollen Ihnen die soeben erklärten Konzepte noch einmal verdeutlichen. Erläuterungen finden Sie oftmals im nachfolgenden Text.

1.3.2 »Testen Sie Ihr Wissen«

Gegen Ende einer Lektion finden Sie meist einen Abschnitt mit Fragen. Hier können Sie testen, wie viel Sie bereits von dem Lernstoff verstanden und behalten haben. Gehen Sie die Fragen gewissenhaft durch und beginnen Sie erst mit der nächsten Lektion, wenn Sie alle Fragen korrekt beantworten

können.

Am Ende des Lernbuchs gibt es zu allen Fragen aller Lektionen die Antworten. Machen Sie sich aber bitte zuerst Gedanken über die Fragen, ehe Sie die Antworten nachschlagen.

1.3.3 »Aufgaben zur Selbstkontrolle«

Mit den »Aufgaben zur Selbstkontrolle« können Sie prüfen, ob Sie den gelernten Stoff wirklich anwenden können. Bearbeiten Sie diese Aufgaben erst, wenn Sie alle Aufgaben im Fließtext bereits erledigt haben. Sie sollten erst dann mit einer neuen Lektion beginnen, wenn Sie alle Aufgaben gelöst und auch verstanden haben.

Die Lösungen zu den Übungsaufgaben jeder Lektion finden Sie im Begleitmaterial.

1.3.4 Optionale Aufgaben

Die optionalen Aufgaben testen ebenfalls den Stoff der jeweiligen Lektion, aber die Aufgaben sind umfassender und meistens auch schwieriger. Die Aufgaben verbinden auch oft den Stoff der aktuellen Lektion mit den Techniken vorheriger Lektionen. Wenn Sie bisher wenig Erfahrung in der Programmierung haben, werden Sie eventuell nicht alle optionalen Aufgaben sofort selbstständig lösen können.

Ist das der Fall, machen Sie sich keine Sorgen und gehen Sie zur nächsten Lektion über. Die Pflichtübungen prüfen alle wichtigen Grundlagen ab. Kehren Sie, nachdem Sie das ganze Buch durchgearbeitet haben, noch einmal zu den optionalen Aufgaben zurück. Sie werden sehen, dass Sie diese nun lösen können.

Es gibt einige Lektionen, in denen die Aufgaben aufeinander aufbauen. Dies gilt jeweils für die Pflichtaufgaben und für die optionalen Aufgaben; wenn Sie also in einer vorangegangenen Lektion nicht alle optionalen Aufgaben gelöst haben, werden Sie in der weiterführenden Lektion zwar alle Pflichtübungen durchführen

können, aber bevor Sie zu den optionalen Aufgaben übergehen, sollten Sie die der vorhergehenden Lektion lösen.

1.4 Anforderungen an PHP

Dieses Lernbuch verwendet *Doctrine 2*, um Daten in einer MySQL-Datenbank abzulegen und wieder auszulesen. Doctrine 2.5 setzt PHP ab Version 5.4 voraus. Sollten Sie eine ältere PHP-Version verwenden, so ist eine Aktualisierung zwingend notwendig. Da jedoch die EOL-Phase von PHP 5.5 am 21.07.2016 endete, gehe ich davon aus, dass Ihnen mindestens eine Entwicklungsumgebung mit **PHP 5.6** zur Verfügung steht. Ich selbst nutze übrigens seit September 2016 eine XAMPP-Installation mit PHP 7.0.

Die meisten Beispiele und Aufgaben wurden ursprünglich auf einem System mit der PHP-Version **5.3.14** entwickelt und getestet. Auf die Behandlung neuer Features aus PHP 5.4, 5.5, 5.6 oder 7.0 habe ich an vielen Stellen bewusst verzichtet, da die Angabe von zwei oder gar drei Varianten bei den Code-Beispielen den Seitenumfang dieses Lernbuchs gesprengt hätte. Der Code ist aber natürlich trotzdem so ausgelegt, dass er mit den neueren PHP-Versionen lauffähig sein sollte.

Ich gehe in allen Beispielen und Musterlösungen mit Datums- bzw. Zeitangaben davon aus, dass in der *php.ini* des verwendeten Webservers bereits die **korrekte Zeitzone** (z. B. `'Europe/Berlin'`) hinterlegt wurde.

Achtung Mac-User: MAMP-Fehler mit PHP 5.6

In Version 3.0 von MAMP ist es möglich, in den Einstellungen eine PHP-Version auszuwählen. In MAMP 3.0.7 ist standardmäßig PHP 5.6 aktiv.

Bei dieser PHP-Version ist jedoch die PDO-Konfiguration fehlerhaft (Meldung `Fatal error: Uncaught exception 'PDOException' with message 'could not find driver'`). Wählen Sie deshalb bitte die vorherige PHP-Version 5.5 aus.

Bei MAMP Version 3.0.4 gab es mit PHP 5.5 wohl das gleiche Problem.

2 Objekt-relationales Mapping

In dieser Lektion lernen Sie
• was objekt-relationales Mapping (ORM) ist. • welche verbreiteten Entwurfsmuster Sie zu diesem Thema kennen sollten.

2.1Das Problem

Sie haben in »Band 1: Grundlagen der OOP« gelernt, welche Vorteile die objektorientierte Programmierung Ihnen bei der Strukturierung Ihres Codes bieten kann. Bisher haben Sie Objekte meist mit jedem Aufruf einer PHP-Seite neu erzeugt und mit Werten befüllt. Das mag zum Kennenlernen der objektorientierten Programmierung in Ordnung sein, aber spätestens, wenn Sie mit echten Benutzereingaben umgehen, müssen Sie diese schließlich irgendwo speichern. Deswegen erfahren Sie nun, wie Sie eine Persistenz in einer relationalen Datenbank wie MySQL umsetzen können.

Hierbei stoßen Sie allerdings auf ein Problem: Relationale Datenbanken speichern ihre Daten nicht in Form von Objekten, sondern in Tabellen ab. Wenn Sie Daten aus einer Datenbank abfragen, erhalten Sie diese normalerweise als Array zurück. Ebenso müssen Sie einen SQL-String bauen, der Ihnen Daten in die Datenbank schreibt, und können nicht direkt das PHP-Objekt ablegen. Kurz gesagt, relationale Datenbanken können nicht mit PHP-Objekten umgehen. Es existieren zwar auch objektorientierte Datenbanken, die aber in Verbindung mit PHP keine Rolle spielen.

Natürlich wollen wir nicht zurück zur alten, prozeduralen Art, Code zu schreiben. Also müssen wir es irgendwie schaffen, objektorientierte Programmierung und relationale Datenbanken zusammenzubringen. Dies nennt man ***objekt-relationales Mapping*** oder kurz ***ORM***.

Als objekt-relationales Mapping bezeichnet man also das Übersetzen der relationalen Welt der Datenbanken in die objektorientierte Welt der

Programmiersprachen, in unserem Fall PHP.

2.2 Verbreitete ORM-Entwurfsmuster

Das Problem, objektorientierte Programmierung und relationale Datenbanken miteinander in Einklang zu bringen, ist nicht neu. Zu diesem Thema wurden schon unzählige Fachartikel und ganze Bücher geschrieben. Viele Softwareentwickler sind sogar der Meinung, das Problem ließe sich überhaupt nicht lösen, da OOP und relationale Datenbanken fundamental unterschiedlich sind. Dieses Problem wurde u.a. schon als das »Vietnam der Softwareentwicklung« bezeichnet.

Dennoch haben sich über die Jahre einige Entwurfsmuster etabliert, die mehr oder weniger gut in der Lage sind, Objekte aus Datensätzen zu bilden und Objekte wieder in die Datenbank zu speichern. Ich möchte Ihnen kurz zwei bekannte Entwurfsmuster vorstellen. Beide Entwurfsmuster entstammen übrigens dem Buch »Patterns of Enterprise Application Architecture« von Martin Fowler (ISBN-13 978-0321127426).

2.2.1 Active Record

Active Record ist momentan das wohl bekannteste der beiden Entwurfsmuster, weil es das Standard-ORM-Pattern in *Ruby on Rails* ist. Die Ihnen bereits aus dem ersten Band bekannten Datenklassen (Datenhaltung) werden um Methoden für den Datenbankzugriff (Datenverwaltung) erweitert. So erhält z. B. die Klasse `Person` eine Methode `insert()`, die ein Objekt dieser Klasse als neuen Datensatz speichert.

Beispiel

```php
<?php

$person = new Person();
$person->setVorname('Arthur');
$person->setNachname('Dent');
$person->insert(); // fuehrt ein SQL-INSERT aus
```

Active Record

***Codebeispiel 1** Active Record*

Der Vorteil dieses Entwurfsmusters ist, dass Sie in PHP wie inzwischen hoffentlich gewohnt mit Klassen und Objekten arbeiten können. Ein einfacher Methodenaufruf auf diese Objekte sorgt für die Persistenz in der Datenbank.

Bereits existierende Datensätze werden über eine sogenannte ***statische Methode*** ausgelesen. Solche statischen Methoden sind direkt an die Klasse gebunden und benötigen keine Instanz, um verwendet zu werden. Erkennbar sind sie an dem Schlüsselwort `static`, welches direkt hinter der Sichtbarkeit angegeben wird. Als Rückgabewert erhalten Sie dann den Datensatz als Objekt der entsprechenden Klasse. Bei mehreren Datensätzen erhalten Sie hingegen meist ein Array von Objekten.

Beispiel

```php
<?php

class Person
{
    public static function find($id)
    {
        // Hier wird der Datensatz mit der ID per SELECT aus
    }
}
```

Model - Active Record find()-Methode

***Codebeispiel 2** Model - Active Record find()-Methode*

```php
<?php

// liefert den Datensatz mit der ID 42 als ein Objekt der Kla
$person = Person::find(42);
echo $person->getVorname(); // gibt den Eintrag der Spalte vo
```

Controller - Active Record find()-Methode

***Codebeispiel 3** Controller - Active Record find()-Methode*

Genaugenommen merken Sie also nicht mehr, wo die gelieferten Daten überhaupt herkommen. Natürlich hat dieser Komfort seinen Preis. Das Active Record-Muster selbst zu programmieren ist relativ aufwendig. Bei größeren Projekten lohnt sich der Aufwand jedoch oft, da Sie die Klassen nur einmal

schreiben, aber oft verwenden werden. Durch die Kombination von Datenhaltung und Datenverwaltung muss man jedoch in Kauf nehmen, dass man mit relativ umfangreichen Klassen arbeitet.

> Bei Active Record repräsentiert eine Klasse immer genau eine Datenbank-Tabelle. Eine Instanz dieser Klasse steht für einen Datensatz dieser Tabelle. Dieses Pattern kombiniert die Datenhaltung mit der Datenverwaltung in einer einzelnen Klasse. Diese Kombination ist das wohl bekannteste Beispiel für einen Verstoß gegen das Single Responsibility Principle, welches wir in »Band 1: Grundlagen der OOP« besprochen haben.

2.2.2 Data Mapper

Das ***Data Mapper***-Entwurfsmuster trennt die Datenhaltung und die Datenverwaltung in zwei separate Klassen. Die eine Klasse enthält die Daten und entspricht den Datenklassen, die Sie bereits kennen. Diese Klasse hat keine Informationen über den für die Datenbank nötigen SQL-Code.

Zu jeder Datenklasse gehört eine sogenannte Mapperklasse, die die Methoden zur Datenverwaltung enthält. Sie arbeiten also wie gewohnt mit Ihren Objekten, und wenn diese in der Datenbank gespeichert werden sollen, übergeben Sie das ganze Objekt dem Mapper und dieser kümmert sich um die Persistierung der Objektdaten.

Beispiel

```php
<?php

$person = new Person();
$person->setVorname('Arthur');
$person->setNachname('Dent');

$mapper = new PersonMapper();
$mapper->insert($person);
```

Data Mapper

Codebeispiel 4 *Data Mapper*

Dieses Entwurfsmuster ist sehr flexibel. Da die Datenklassen keine Ahnung

haben, wohin ihre Objekte gespeichert werden, ist es mit diesem Muster beispielsweise sehr einfach, das Daten-Backend komplett auszutauschen.

Wie üblich geht diese Flexibilität mit einem erhöhten Programmieraufwand einher. Ein Data Mapper ist sehr aufwendig zu implementieren, spielt seine Stärken aber v.a. bei sehr umfangreichen und komplexen Projekten aus.

> Beim Data Mapper trennt man die Funktionalität der Datenhaltung von der Datenverwaltung. Dies geschieht durch die Trennung der Funktionalität in zwei Klassen. Die Mapperklasse repräsentiert immer die Tabelle, die Datenklasse die einzelnen Datensätze. Ein Objekt der Datenklasse entspricht normalerweise einem Datensatz der Tabelle.

2.3 Zusammenfassung

Sie haben in dieser Lektion eine kurze Übersicht über zwei gängige Entwurfsmuster zum Verwalten von relationalen Datenbanken in der objektorientierten Programmierung kennengelernt. In den folgenden Lektionen werden wir uns mit ***Doctrine 2*** beschäftigen, einer sehr mächtigen Umsetzung eines Data Mappers. Der Vorgänger ***Doctrine 1*** war übrigens noch eine Umsetzung auf Basis von ***Active Record***.

2.4 Testen Sie Ihr Wissen

1. Wie viele Klassen repräsentieren normalerweise eine Datenbank-Tabelle, wenn Sie Active Record verwenden?

2. Wie viele Objekte erhalten Sie in PHP, wenn Sie mit Active Record 30 Datensätze aus der DB auslesen?

3. Wie viele Klassen repräsentieren normalerweise eine Datenbank-Tabelle, wenn Sie Data Mapper verwenden?

4. Wie viele Objekte erhalten Sie in PHP, wenn Sie mit dem Data Mapper 30

Datensätze aus der DB auslesen?

3 Composer, Packagist & Co.

In dieser Lektion lernen Sie
• was das Werkzeug Composer leisten kann und wie man es benutzt. • was Packagist ist und wofür Sie es benötigen. • was ein Autoloader ist und was er Ihnen erspart.

3.1Einleitung

Von ***Active Record*** und dem ***Data Mapper***-Entwurfsmuster kennen Sie inzwischen die Ansätze für zwei gängige Varianten, die einen objektorientierten Zugriff auf relationale Datenbanken ermöglichen. Nun wird es Zeit, dass wir uns eines dieser Muster, nämlich den Data Mapper am Beispiel von *Doctrine*, genauer ansehen.

Doch dabei ergibt sich ein Problem: Bei Doctrine handelt es sich nämlich gleich um mehrere Bibliotheken (Code-Sammlungen) eines Drittanbieters, und diese müssen Sie irgendwie auf Ihrem Entwicklungsrechner installieren. Das Problem wächst noch dadurch, dass Sie nicht nur die drei Bibliotheken Common, DBAL (Database Abstraction Layer) und ORM (Object Relational Mapper) aus dem Doctrine-Projekt benötigen, die allein schon mehr als 500 Dateien umfassen.

Zusätzlich erscheinen regelmäßig neue Versionen aller Bibliotheken, wodurch sich Abhängigkeiten zwischen den Bibliotheken ergeben, sofern einzelne Features aufeinander aufbauen. Viele Bibliotheken sind nämlich abhängig von anderen Bibliotheken (z. B. ORM von Common und DBAL). Damit sich also keine Probleme daraus ergeben, dass einzelne Bibliotheken nicht miteinander harmonieren, sollten Sie mit aufeinander abgestimmten Versionen arbeiten.

Doch bevor wir uns mit der Lösung dieser Probleme beschäftigen, möchte ich einmal kurz die drei Bibliotheken Common, DBAL und ORM voneinander abgrenzen:

- Die ***Common***-Bibliothek ist die Basis von Doctrine. Sie enthält

beispielsweise Hilfsmittel, um ein Debugging von Doctrine-Objekten vornehmen zu können. Als kleine Warnung will ich schon jetzt erwähnen, dass Doctrine-Objekte bei der Nutzung von `var_dump()` öfters Probleme machen.

- Die ***DBAL***-Bibliothek bietet eine leistungsstarke Datenbankabstraktionsschicht und ermöglicht so einen einheitlichen Zugriff auf bekannte Datenbank Management Systeme. Doctrine unterstützt derzeit die DBMS MySQL, Oracle, PostgreSQL, SQLite, SAP SQL Anywhere und Microsoft SQL Server. Die Bibliothek setzt hierbei auf PDO (PHP Data Objects) auf, welche seit PHP 5.1 ein fester Bestandteil der Sprache sind und selbst schon eine Abstraktionsebene für den Datenbankzugriff darstellen. DBAL nimmt jedoch noch einige Verbesserungen an dem einheitlichen SQL-basierten Zugang von PDO vor.
- Die ***ORM***-Bibliothek ist der eigentliche Data Mapper und ermöglicht den objektorientierten Datenbank-Zugriff für alle unterstützten DBMS. Eines der Haupt-Features ist hierbei der objektorientierte SQL-Dialekt DQL. Mit diesem werden wir uns später noch intensiv befassen.

3.2 Composer-Einführung

Bevor wir uns wirklich mit Doctrine beschäftigen können, müssen wir uns zunächst mit *Composer* (dt. Komponist, also jemand, der zusammenstellt bzw. arrangiert) befassen. Bei Composer handelt es sich um ein noch relativ junges Open-Source-Projekt, welches uns ein Werkzeug zum Managen der Abhängigkeiten eines PHP-Projekts zur Verfügung stellt.[1] Wir werden im weiteren Verlauf Composer verwenden, um die benötigten Bibliotheken und deren Abhängigkeiten zu installieren.

---

1 Vorbilder von Composer sind NPM (https://www.npmjs.com/) und Bundler (http://bundler.io/).

3.2.1 composer.phar

Das Werkzeug *Composer* ist in Form einer einzigen Phar-Datei verfügbar. Phar bzw. PHP Archive ist eine Erweiterung in PHP, die es ermöglicht, aus einer

komprimierten Archivdatei heraus Programme oder Dateien zu verarbeiten. Diese Phar-Datei kann im Konsolen-Fenster Ihres Betriebssystems (auch Kommandozeile oder Terminal genannt) benutzt werden, um eine Reihe von Aktionen für Sie zu erledigen. Die wichtigsten Befehle für diese Aktionen werden Sie im Rahmen dieses Lernbuchs kennenlernen.[2]

---

2 Eine Übersicht mit einer kurzen Erklärung jedes Befehls finden Sie unter http://composer.json.jolicode.com/.

Achtung: Windows-Pfadprobleme

In der Konsole kann man ein Programm (z. B. *php.exe*) normalerweise nur starten, wenn man sich in der Konsole bereits im Ordner mit dem Programm befindet oder beim Aufruf den genauen Pfad zum Programm nennt.

Unter Windows mit standardmäßig installiertem XAMPP kann man also beispielsweise zunächst mit `cd \xampp\php` in den passenden Ordner auf Laufwerk *c:* wechseln und dann `php.exe -v` aufrufen, um die PHP-Version zu erfahren. Alternativ kann man dies aber auch mit `c:\xampp\php\php.exe -v` aus jedem beliebigem Ordner machen.

Das ist beides jedoch relativ umständlich. Deswegen kann man den Pfad weglassen, sofern das Betriebssystem diesen bereits kennt. Mit dem Befehl `path` bekommt man alle bekannten Pfade angezeigt. Taucht der Pfad *c:\xampp\php* in dieser Liste nicht auf, so kann man ihn entweder ergänzen oder man verwendet alternativ die **Shell** von XAMPP. Diese findet man im *XAMPP Control Panel* rechts oben.

Spätestens jetzt sollten Sie in einem beliebigen Ordner `php -v` eingeben und hierbei auf den Pfad und die Dateiendung komplett verzichten können.

3.2.2 Die Projekt-Struktur

Mit der Zeit gewöhnt sich jeder PHP-Entwickler eine weitestgehend gleichbleibende Projekt-Struktur an, die er bei jedem Projekt verwendet und lediglich mit anderen Inhalten befüllt. Eine solche Projekt-Struktur habe ich

basierend auf der finalen MVC-Struktur aus »Band 1: Grundlagen der OOP« bereits für Sie vorbereitet. Doch wie bekommen Sie diese Struktur auf Ihren Entwicklungsrechner? Auch hierbei hilft Ihnen Composer, wie Sie gleich sehen werden.

Übung 1:

1. Öffnen Sie in Ihrem bevorzugten Dateimanager das Document Root Ihres Webservers. Bei XAMPP (Windows) ist dies *C:\xampp\htdocs* und bei MAMP (Mac OS X) */Applications/MAMP/htdocs*. Kopieren Sie die Phar-Datei *composer.phar* aus dem Begleitmaterial in diesen Ordner.
2. Öffnen Sie nun ein Konsolen-Fenster und wechseln Sie dort ebenfalls in das Document Root. Ein solcher Verzeichniswechsel geschieht üblicherweise mit dem Befehl `cd`. Überprüfen Sie mit dem Composer-Kommando `php composer.phar about`, ob Sie bisher alles richtig gemacht haben. Es sollte hierdurch ein kurzer Text über Composer im Konsolen-Fenster ausgegeben werden. Dies war Ihr erstes Composer-Kommando. Alle weiteren Composer-Kommandos beginnen ebenfalls mit der Angabe `php composer.phar`.
3. Verwenden Sie nun das Konsolenkommando `php composer.phar self-update`, um Composer auf die neueste Version zu aktualisieren. Alternativ finden Sie den aktuellsten Snapshot der *composer.phar* auch unter http://getcomposer.org/composer.phar.
4. Probieren Sie nun das Konsolenkommando `php composer.phar create-project webmasters/doctrine-skeleton:1.0.0` aus.
5. Probieren Sie abschließend das Konsolenkommando `php composer.phar --no-install create-project webmasters/doctrine-skeleton:1.0.0 newsticker_d2` aus.

Kommen wir kurz zum Aufbau der letzten beiden Kommandos und danach zu deren Ergebnis. Wir sagen PHP, dass es die Anwendung in der Datei *composer.phar* ausführen soll. Dieser Anwendung geben wir wiederum das Kommando `create-project`. Composer soll also ein Projekt bzw. genauer

gesagt eine Projekt-Struktur erstellen. Als Quelle soll hierbei das Paket `webmasters/doctrine-skeleton` in der Version `1.0.0` verwendet werden.

Doch was bedeutet die Angabe `--no-install` und wieso geben wir in der zweiten Variante des Kommandos am Ende noch `newsticker_d2` an? Diese optionalen Angaben beeinflussen das Ergebnis des Kommandos.

Schauen Sie sich das Document Root Ihres Webservers nun etwas genauer an. Sie sollten dort zwei neue Ordner *doctrine-skeleton* und *newsticker_d2* finden. Der Ordner *doctrine-skeleton* entstand durch die erste Variante des Kommandos und der andere durch die zweite. Wenn man also nicht explizit einen speziellen Ordner für das Projekt angibt, so benutzt Composer als Standard den zweiten Teil des Paket-Namens. Schauen wir uns nun den Inhalt des Ordners *newsticker_d2* an, wobei ich nur die derzeit relevanten Unterordner und Dateien angegeben habe:

```
-- config
-- css
-- inc
-- src
---- Controllers
---- Entities
---- Repositories
-- templates
index.php
composer.json
```

Verzeichnisansicht

Codebeispiel 5 *Verzeichnisansicht*

Vergleicht man diese Projekt-Struktur mit dem Inhalt des Ordners *doctrine-skeleton*, so fällt auf, dass es dort einen zusätzlichen Ordner *vendor* gibt. Mit der Option `--no-install` haben wir also verboten, dass dieser angelegt und befüllt wird.

Hier noch einmal der grundsätzliche Aufbau des Kommandos `create-project`, wobei das Kommando natürlich noch weitere Optionen kennt:

```
php composer.phar --option1 --option2 create-project paket:vers
```

Übung 2:

Verschieben Sie die *composer.phar* aus dem Document Root in unseren Projekt-Ordner *newsticker_d2*. Von jetzt an müssen Sie für alle weiteren Konsolenkommandos von Composer **immer** zunächst in den Projekt-Ordner wechseln und können dann erst das Kommando aufrufen.

Hinweis

Es besteht die Möglichkeit, Composer unter Windows und auf Unix-basierten Systemen auch global zu installieren. Die Composer-Kommandos beginnen dann anstatt mit `php composer.phar` meist einfach nur mit `composer`.

3.3 Composer & Packagist

Doch drei wichtige Fragen wurden noch nicht beantwortet:

1. Woher hat Composer die Dateien für die Projekt-Struktur bekommen?
2. Was hat es mit dem Ordner *vendor* auf sich?
3. Wann kommen wir endlich zu Doctrine?

Kurz und bündig würden die Antworten ***Packagist***, **Standard-Ordner** für Bibliotheken und **bald** lauten, doch jeden dieser Punkte müssen wir nun noch etwas ausführlicher beleuchten.

3.3.1 composer.json

Betrachten wir zunächst die Frage zum Standard-Ordner *vendor* und sehen uns hierfür die Datei *composer.json* an.

Beispiel

```
{
    "autoload": {
        "psr-0": {
            "Controllers": "./src/",
            "Entities": "./src/",
            "Repositories": "./src/"
        }
    },
    "require": {
        "php": ">=5.4",
        "doctrine/orm": "2.5.5"
    }
}
```

composer.json

Codebeispiel 6 *composer.json*

Die projektspezifische Datei *composer.json* erlaubt es dem Entwickler unter anderem, im Abschnitt `require` die Bibliotheken inklusive der gewünschten Version zu definieren, die die Anwendung benötigt. Im Gegensatz zu beispielsweise PEAR arbeitet Composer somit auf Basis eines bestimmten Projektes und nicht systemglobal.

Bei der Definition von Abhängigkeiten (engl. ***dependencies***) muss für jede Bibliothek (bei Composer Paket genannt) der Paketname, bestehend aus Vendor (dt. Anbieter) und dem eigentlichen Namen der Bibliothek (der Projektname), angegeben werden. Auch Projekt-Strukturen sind im Sinne von Composer Pakete, werden jedoch nicht im Standard-Ordner *vendor* installiert. Da oftmals identische Projektnamen existieren, verhindert die Angabe des Anbieters Probleme bei der Identifizierung der benötigten Bibliotheken. Suchen Sie beispielsweise einmal mit einer Suchmaschine Ihrer Wahl nach **Doctrine Extensions**.

Für jedes Paket muss zudem die gewünschte Version hinterlegt werden. Dabei sind nachfolgende Varianten möglich.

1. `"2.5.5"`

   In diesem Fall haben wir die eindeutige Versionsnummer eines Pakets angegeben und Composer erlaubt deswegen keine Abweichung in der Version. Allerdings werden durchaus abweichende Schreibweisen, wie beispielsweise "v2.5.5" automatisch berücksichtigt.

2. `">=2.5.5"`

   Hier geben wir einen nach oben offenen Bereich vor. Die Version muss mindestens "2.5.5" sein. Es ist jedoch auch der Sprung auf "2.6.0" (Minor) oder gar "3.0.0" (Major) möglich. Auch alle sonstigen Versionen bzw. Versions-Sprünge ab der Mindestversion sind natürlich erlaubt (z. B. "3.0.1", "4.0.0" oder "10.4.6").

3. `">=2.5.5,<2.7"`

   In diesem Fall ist der Bereich nach unten und nach oben beschränkt. Die Version muss mindestens "2.5.5" sein. Ein Minor-Sprung auf "2.6.0" ist möglich, ein Minor-Sprung auf "2.7.0" oder gar ein Major-Sprung auf "3.0.0" jedoch ausgeschlossen.

4. `"~2.5.5"`

   Mit der **Tilde** am Anfang sagen wir Composer, dass wir eine »ungefähre« Version ab "2.5.5" wünschen. Ein höherer Patch-Level (Angabe hinter dem letzten Punkt) ist erlaubt, ein Sprung auf den Minor-Release "2.6.0" jedoch nicht möglich. "~2.5.5" ist equivalent zu ">=2.5.5,<2.6.0", während "~2.5" equivalent zu ">=2.5.0,<3.0.0" ist.

5. `"2.5.*"`

   Wir benutzen einen **Platzhalter** (engl. Wildcard) anstatt der Angabe eines genauen Patch-Levels. Alle Versionen inklusive "2.5.0" sind möglich, sofern sich lediglich der Patch-Level ändert. "2.5.*" ist somit equivalent zu ">=2.5.0,<2.6.0". Alternativ ist natürlich auch die Versionsangabe "2.*" (">=2.0.0,<3.0.0") und theoretisch sogar die Angabe "*" möglich.

6. `"^2.5.5"`

   Das Composer-Verhalten beim **Auslassungszeichen** (engl. Caret) am Anfang erklärt man am besten mittels Beispielen:

   "^2.5.5" ist equivalent zu ">=2.5.5,<3.0.0". Es wird also eine Minimalversion "2.5.5" festgelegt, höhere Patch-Level und Minor-Sprünge sind erlaubt, die nächste Major-Version "3.0.0" ist jedoch ausgeschlossen. Wenn sich ein Projekt an Semver hält, d.h. Patch-Level ist ein abwärtskompatibler Bugfix, Minor ist eine abwärtskompatible Feature-Ergänzung und Major ist eine nicht zwingend abwärtskompatible Änderung,

so sind somit alle Versionssprünge erlaubt, bei denen der Code abwärtskompatibel bleiben müsste.

"^2.5" ist übrigens equivalent zu ">=2.5.0,<3.0.0" und somit zu "~2.5", die nächste Major-Version "3.0.0" ist also ausgeschlossen.

Bei Versionsnummern unter "1.0" ändert sich das Verhalten beim **Auslassungszeichen**. "^0.3" ist equivalent zu ">=0.3.0,<0.4.0". Hier sind also nur Sprünge im Bereich des Patch-Levels erlaubt.

7. `"2.0.0 - 3.0.0"`
   In diesem Fall muss die Version mindestens "2.0.0" sein. Sprünge der Version bis inklusive 3.0.0 sind möglich. Es handelt sich also um eine Bereichsangabe, bei der Anfang und Ende inklusive sind.

   Etwas verwirrend wird die Bereichsangabe mit dem **Bindestrich**, wenn man unvollständige Versionsangaben macht (z. B. ohne Patch-Level). So ist "2.0 - 3.0" equivalent zu ">=2.0.0,<3.1.0", da für den fehlenden Patch-Level ein Platzhalter verwendet wird und somit nicht nur "3.0.0" sondern auch alle weiteren Patch-Level erlaubt sind.

Innerhalb der erlaubten Versionen versucht Composer dann, eine passende und möglichst aktuelle Version zu finden. Im Rahmen des Beispielprojekts dieses Lernbuchs werde ich jedoch immer mit eindeutigen Versionsangaben arbeiten, um Probleme mit ungetesteten Versionen bzw. Kombinationen zu vermeiden.

3.3.2 Packagist

Zusätzlich zum Paket und der Version ist noch die Paket-Installationsquelle relevant. Die Standard-Quelle von Composer ist Packagist. Für Pakete, die dort vorhanden sind, muss die Herkunft nicht explizit angegeben werden.

Meines Wissens hostet *Packagist* jedoch keine Dateien selbst, sondern verwaltet lediglich registrierte Pakete und kennt für diese den eigentlichen Hoster. Zur Erforschung der verfügbaren Pakete gibt es zwei Möglichkeiten. Die Website selbst bietet eine Suche und mit https://packagist.org/explore/ eine spezielle Einstiegsseite, wo ständig neue und aktualisierte Pakete beworben

werden. Eine Suche ist jedoch auch in der Konsole möglich.

Übung 3:

1. Testen Sie mit dem Kommando `php composer.phar search webmasters`, ob die Suche das Paket `webmasters/doctrine-skeleton` findet.
2. Finden Sie nun mit dem Kommando `php composer.phar browse webmasters/doctrine-skeleton` heraus, wo das Paket gehostet ist. Sollte sich hierdurch nicht automatisch eine Website im Browser öffnen, können Sie alternativ natürlich auch die Suche auf der Website von Packagist verwenden.[3]

---

3 Die entsprechende Angabe auf der Detailseite der Suche hat den Tooltip **Canonical Repository URL**.

Für Abhängigkeiten, die (noch) nicht über *Packagist* verfügbar sind, müssen Sie eine Quelle angeben, aus der das Paket geladen werden soll. Ich habe mich jedoch auf verfügbare Pakete beschränkt, um den Aufwand zu minimieren.[4]

---

4 Alle diese Pakete werden auf *GitHub* gehostet.

3.3.3 Die eigentliche Installation

Im Ordner *newsticker_d2* hatten wir ja mittels einer Option die automatische Installation der Abhängigkeiten verboten. Wie können Sie diese nun nachholen? Das Konsolenkommando zur (ersten) Installation der in der *composer.json* definierten Pakete sieht folgendermaßen aus:

```
php composer.phar install
```

Composer installiert dabei in das Standard-Verzeichnis *vendor*. In diesem Verzeichnis liegen dann alle Abhängigkeiten sortiert nach Anbietern. Natürlich nur sofern PHP ausreichende Berechtigungen hierfür hatte und den Befehl erfolgreich ausführen konnte.

```
Loading composer repositories with package information
Installing dependencies (including require-dev)
- Installing doctrine/lexer
- Installing doctrine/inflector
- Installing doctrine/collections
- Installing doctrine/cache
- Installing doctrine/annotations
- Installing doctrine/common
- Installing symfony/console
- Installing doctrine/dbal
- Installing doctrine/orm
Writing lock file
Generating autoload files
```

gekürzte Composer-Ausgabe im Konsolen-Fenster

***Codebeispiel 7** gekürzte Composer-Ausgabe im Konsolen-Fenster*

Composer bezieht bei der Installation alle Pakete und löst dabei auch Abhängigkeiten der Bibliotheken (ORM) von weiteren Bibliotheken (z. B. Common und DBAL) auf. Da das Paket `doctrine/dbal` im Paket `doctrine/orm` als Abhängigkeiten hinterlegt ist und das Paket `doctrine/common` wiederum eine Abhängigkeit von `doctrine/dbal` ist, können Sie auf die Angabe dieser Pakete in Ihrer *composer.json* verzichten.

Wie jede andere PHP-Datei können die so installierten Bibliotheken danach mittels FTP-Client vom Entwicklungsrechner auf Ihren Webspace oder Webserver hochgeladen werden.

Übung 4:

1. Öffnen Sie die Datei *newsticker_d2/templates/layout.tpl.php* und passen Sie den Inhalt des `title`- und `h1`-Tags an. Wir wollen nun schließlich an einem `Newsticker` arbeiten.
2. Öffnen Sie ein Konsolen-Fenster und wechseln Sie dort in den Ordner *newsticker_d2*.
3. Installieren Sie nun mit dem Konsolenkommando `php composer.phar install` die benötigten Pakete und erforschen Sie ein wenig den Inhalt des Verzeichnisses *vendor*.

Im *vendor*-Ordner sollten Sie vier Verzeichnisse vorgefunden haben, nämlich *bin*, *composer*, *doctrine* und *symfony*. Die ersten beiden Ordner dienen eher Verwaltungsaufgaben und die restlichen beiden Ordner enthalten jeweils die Bibliotheken eines bestimmten Anbieters.

3.4 Wichtige Composer-Dateien

Im Rahmen der Installation erstellt Composer zwei für uns wichtige Dateien, nämlich die Datei *composer.lock* direkt im Projekt-Verzeichnis (Meldung »Writing lock file«) und die Datei *vendor/composer/autoload_namespaces.php* (Meldung »Generating autoload files«).

3.4.1 composer.lock

In der *composer.lock* speichert Composer die genauen Versionen aller installierten Pakete, was vor allem bei Abhängigkeiten von Abhängigkeiten und der Verwendung von Platzhaltern und Versionsbereichen nützlich ist.

> Sollten Sie mit mehreren Entwicklern an einem Projekt arbeiten, so müssen Sie nicht den kompletten Inhalt des Verzeichnisses *vendor* kopieren. Es reicht eine Kopie der *composer.lock* und danach eine Installation der Bibliotheken mittels `php composer.phar install`. So erhalten Sie auf allen Entwicklungsrechnern identische Versionen der einzelnen Bibliotheken. Diese Vorgehensweise ist natürlich grundsätzlich auch auf einem Produktivsystem möglich, sofern Sie dort über einen Konsolenzugriff verfügen.

3.4.2 autoload_namespaces.php

Wie schon erwähnt, haben Sie allein mit Doctrine über 500 Dateien auf Ihrem Entwicklungsrechner installiert. Wenn Sie für alle diese Dateien eine Einbindung mittels `require` oder `require_once` vornehmen müssten, so wären Sie eine ganze Weile beschäftigt. Und dabei ist noch nicht einmal berücksichtigt, dass Sie gar nicht wissen, welche der dortigen Klassen Sie überhaupt benötigen und

welche nicht.

Seit PHP 5 ist dies aber auch gar nicht länger notwendig. Man kann eine sogenannte `__autoload()`-Funktion definieren, die automatisch aufgerufen wird, wenn man eine noch nicht eingebundene Klasse benutzt. Durch den Aufruf dieser Funktion unternimmt PHP einen letzten Versuch, die Klasse zu laden, bevor es zu einer Fehlermeldung kommt. Selbst auf einem produktiven Webserver ist ein Autoloading übrigens durchaus empfehlenswert.

Erfreulicherweise müssen wir den ***Autoloader*** nicht selber schreiben, dieser wird uns von Composer zur Verfügung gestellt. In der Datei *autoload_namespaces.php* speichert Composer bei jeder Installation, wo die unterschiedlichen Pakete abgelegt sind, und ermöglicht damit die Funktionsfähigkeit des Autoloaders.

Beispiel

```php
<?php

// autoload_namespaces.php @generated by Composer

$vendorDir = dirname(dirname(__FILE__));
$baseDir = dirname($vendorDir);

return array(
    'Symfony\\Component\\Console\\' => array($vendorDir . '/s
    'Repositories' => array($baseDir . '/src'),
    'Entities' => array($baseDir . '/src'),
    'Doctrine\\ORM\\' => array($vendorDir . '/doctrine/orm/li
    'Doctrine\\DBAL\\' => array($vendorDir . '/doctrine/dbal/
    'Doctrine\\Common\\Lexer\\' => array($vendorDir . '/docti
    'Doctrine\\Common\\Inflector\\' => array($vendorDir . '/d
    'Doctrine\\Common\\Collections\\' => array($vendorDir .
    'Doctrine\\Common\\Cache\\' => array($vendorDir . '/docti
    'Doctrine\\Common\\Annotations\\' => array($vendorDir .
    'Doctrine\\Common\\' => array($vendorDir . '/doctrine/com
    'Controllers' => array($baseDir . '/src'),
);
```

autoload_namespaces.php

Codebeispiel 8 *autoload_namespaces.php*

Die Array-Schlüssel sind umgekehrt alphabetisch sortiert und Sie sollten hier

drei wichtige Einträge finden können, nämlich jene mit den Schlüsseln `'Repositories'`, `'Entities'` und `'Controllers'`. Für jede dieser drei Sorten von Klassen wird grundsätzlich festgelegt, dass die zugehörigen Dateien der Klassendefinitionen im Ordner *src* zu finden sind.

Damit ist zwar alles für ein Autoloading unserer Abhängigkeiten vorbereitet, zur eigentlichen Nutzung fehlt uns allerdings noch etwas. Was dies ist, erfahren Sie in Lektion 4 und Lektion 5. Bitte haben Sie bis dahin Geduld.

3.5 Zusammenfassung

Ein Abhängigkeits-Manager wie Composer löst für Sie eine ganze Reihe von Problemen:

- Eine automatische Installation aller benötigten Bibliotheken auf Basis einer kurzen Konfigurationsdatei.
- Eine Auflösung von Paket-Abhängigkeiten. So brauchen Sie nur `doctrine/orm` hinzuzufügen und bekommen `doctrine/dbal` und `doctrine/common` automatisch dazu.
- Sie erhalten sogar einen Autoloader, der nur wenig manuelle Konfiguration benötigt.

Ganz nebenbei erhalten Sie mit einem einzigen Befehl auch noch eine fertige Projekt-Struktur, die Sie nur noch mit Inhalten füllen müssen.

3.6 Testen Sie Ihr Wissen

1. Welche Konsolenkommandos von Composer haben Sie kennengelernt?
2. Was erspart Ihnen eine `__autoload()`-Funktion?

4 Doctrine-Entities

In dieser Lektion lernen Sie
• wie Sie Datenklassen für Doctrine anpassen müssen. • wie Sie Annotationen zur Konfiguration von Doctrine nutzen.

4.1Einleitung

Sie haben nun eine Projekt-Struktur, und die Doctrine-Pakete in dieser installiert. Doch bevor Sie wirklich mit Doctrine arbeiten können, benötigen Sie auf jeden Fall mindestens eine Datenklasse, die für Doctrine vorbereitet ist.

4.2 Die Beispiel-Datenbank

Für die Implementierung von Doctrine möchte ich auf das Anwendungsbeispiel eines einfachen **Newstickers** zurückgreifen.

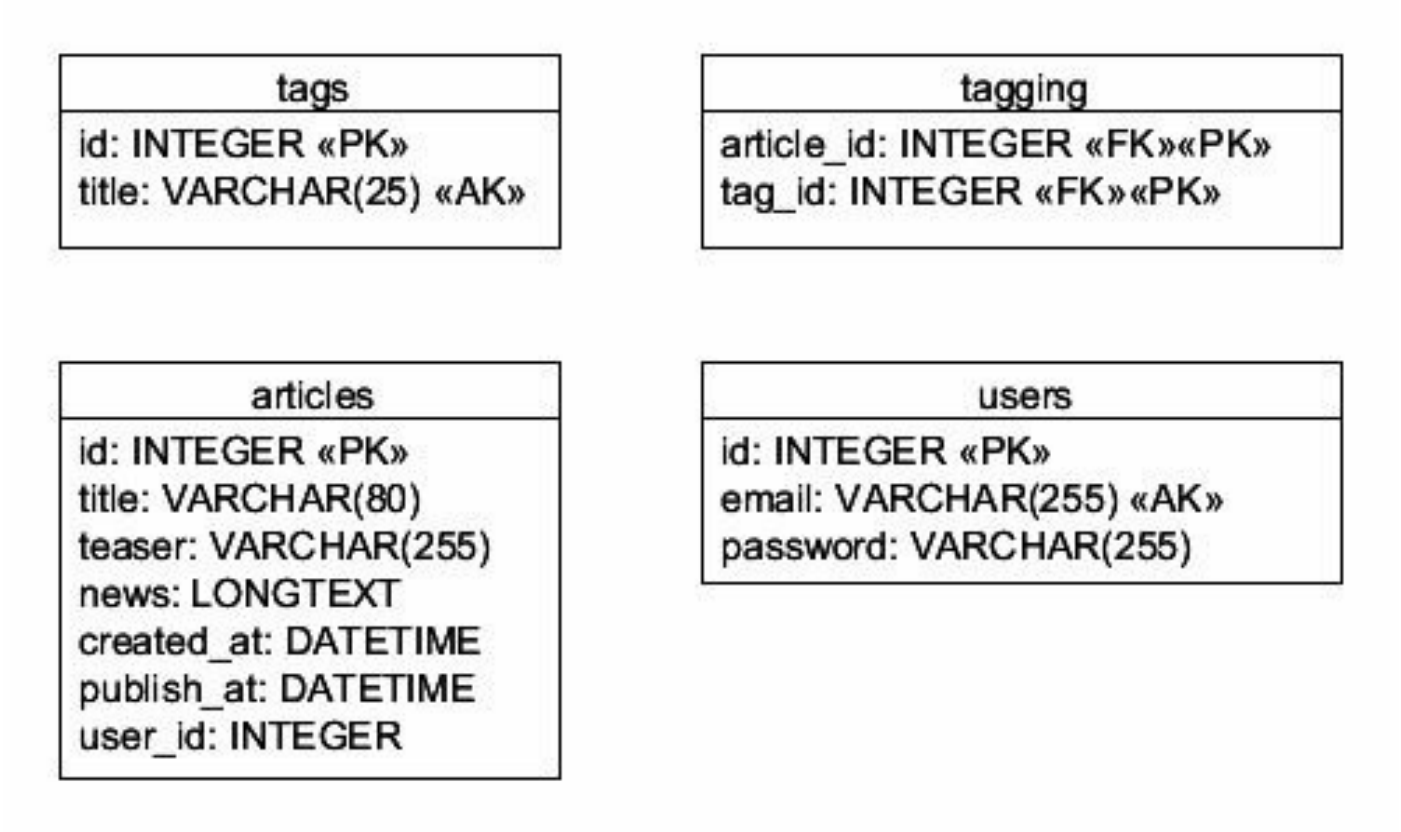

***Abb. 1** Physisches Datenmodell der Newsticker-Datenbank*

Wie es in den meisten größeren Projekten üblich ist, verwenden wir fortan englische Bezeichner. Alle Beispiele der folgenden Lektionen beziehen sich auf diese Datenbank.

4.2.1 Die Datenklasse Tag

Beginnen wir mit der Tabelle `tags` und erstellen in der Datei *Tag.php* eine Klasse `Tag` mit Gettern und Settern. Diese Klasse gehört in den Ordner *src/Entities*. Überfliegen Sie die Datenklasse erst einmal nur. Die Besonderheiten werde ich Ihnen gleich erläutern.

Beispiel

```php
<?php

class Tag
{
    protected $id = 0;
    protected $title = '';

    public function __construct(array $data = array())
```

```
 9      {
10          $this->setData($data);
11      }
12
13      public function  __toString()
14      {
15          return $this->getTitle();
16      }
17
18      public function setData(array $data)
19      {
20          if ($data) {
21              foreach ($data as $k => $v) {
22                  $setterName = 'set' . ucfirst($k);
23                  if (method_exists($this, $setterName)) {
24                      $this->$setterName($v);
25                  }
26              }
27          }
28      }
29
30      /* *** Getter and Setter *** */
31
32      public function getId()
33      {
34          return $this->id;
35      }
36
37      public function getTitle()
38      {
39          return $this->title;
40      }
41
42      public function setTitle($title)
43      {
44          $this->title = $title;
45      }
46  }
```

src/Entities/Tag.php (Version 1)

Codebeispiel 9 *src/Entities/Tag.php (Version 1)*

Die Klasse verwaltet die Attribute `id` und `title` mit den entsprechenden Zugriffsmethoden. Das erste Attribut ist neu und wir verwenden es, um einen Datensatz in der Datenbank eindeutig identifizieren zu können (Stichwort ***Primärschlüssel***).

Methode __construct()

Da wir mit englischen Attributs-Bezeichnern arbeiten, benutzen wir natürlich auch englische Variablen-Bezeichner in den Methoden. Ansonsten hat sich gegenüber »Band 1: Grundlagen der OOP« nichts geändert.

Setter für ID fehlt

Nein, der Setter für die ID wurde nicht vergessen, er fehlt mit voller Absicht! Wir werden uns beim Erzeugen der ID auf die Autoinkrement-Funktionalität von MySQL verlassen, also darf die ID nicht von PHP aus verändert werden.

Methode __toString()

Die Methode `__toString()` gibt das Attribut aus, das das Objekt am besten beschreibt, im Fall von `Tag` also `$title`. So können Sie einfach `echo` ohne Getter verwenden, um eine aussagekräftige Information über ein `Tag`-Objekt zu erhalten.

4.2.2 Namespaces

Ähnlich wie Verzeichnisse in einem Betriebssystem bieten Namespaces seit PHP 5.3 eine Möglichkeit zur Kapselung (bzw. Gruppierung) von Klassen. Um das Beispiel von php.net zu verwenden: In einem Betriebssystem kann die Datei *foo.txt* unter diesem Dateinamen im gleichen Verzeichnis nur ein einziges Mal existieren. Wollen Sie eine zweite Datei unter dem identischen Namen speichern, so müssen Sie dies in einem anderen Verzeichnis tun. Um auf die Datei *foo.txt* von außerhalb des jeweiligen Verzeichnisses zuzugreifen, müssen Sie dem Dateinamen den Verzeichnisnamen und ein Trennzeichen voranstellen. Genauso verhält es sich bei Klassen. Es kann immer nur eine Klasse mit einem bestimmten Namen geben. Benötigt man einen bestimmten Klassennamen erneut, so muss man eine Unterscheidung durch einen vorangestellten Zusatz ermöglichen. Diese Zusätze sind die sogenannten Namespaces.[5]

---

5 Für ein ausführliches Beispiel auf Basis von Telefonnummern möchte ich Sie auf den Beitrag http://goo.gl/xMxxty im Online-Magazin SHIFT verweisen. Auch Postleitzahlen sind übrigens mit Namespaces vergleichbar.

In PHP wurden Namespaces eingeführt, um Probleme zu lösen, auf die man bei der Benutzung von wiederverwendbaren Code-Elementen stößt:

- Namenskollisionen zwischen eigenem und fremdem Code (beispielsweise internen PHP-Klassen oder Klassen von Drittanbietern).
- Vermeidung von `Extrem_Langen_Namen` (Einführung von Kurznamen und Aliasen) zur Verbesserung der Lesbarkeit des Quellcodes.

> Die Namespaces `PHP` und `php`, sowie zusammengesetzte Namen, die mit diesen Bezeichnern beginnen (wie z. B. `PHP\Entities`), sind zur internen Verwendung in der Sprache reserviert und sollten im eigenen Code nicht verwendet werden.

Beispiel

```php
<?php

namespace Entities;

class Tag
{
    protected $id = 0;
    protected $title = '';

    public function __construct(array $data = array())
    {
        $this->setData($data);
    }

    public function  __toString()
    {
        return $this->getTitle();
    }

    public function setData(array $data)
    {
        if ($data) {
            foreach ($data as $k => $v) {
                $setterName = 'set' . ucfirst($k);
                if (method_exists($this, $setterName)) {
                    $this->$setterName($v);
                }
            }
```

```
29          }
30      }
31
32      /* *** Getter and Setter *** */
33
34      public function getId()
35      {
36          return $this->id;
37      }
38
39      public function getTitle()
40      {
41          return $this->title;
42      }
43
44      public function setTitle($title)
45      {
46          $this->title = $title;
47      }
48  }
```

src/Entities/Tag.php (Version 2 mit Namespaces)

Codebeispiel 10 *src/Entities/Tag.php (Version 2 mit Namespaces)*

In den Datenklassen werde ich fortan überall den Namespace `Entities` verwenden. Die Angabe dieses Namespaces sollte als erste Anweisung in der Datei der Klasse erfolgen. Jeder Namespace entspricht bei uns 1:1 einem Unterordner von *src*, d.h. unsere Controller-Klassen verwenden fortan auch einen Namespace, und zwar `Controllers`.

> Datenklassen werden im Ordner *src/Entities* abgelegt, und der Dateiname verwendet die gleiche Schreibweise wie der Name der Klasse. Die Klasse `Tag` mit dem Namespace `Entities` gehört also in die Datei *src/Entities/Tag.php*.
>
> Sie könnten übrigens theoretisch eine beliebige Anzahl von `Tag`-Klassen in Ihrem Projekt haben, sofern jede davon einen anderen Namespace und somit einen anderen Unterordner verwendet.

Überlegen wir nun, wie wir die geänderte Klasse in einem Controller ansprechen müssen. Als Beispiel verwende ich einen Mini-Controller, der lediglich eine leere Instanz unserer `Tag`-Klasse anlegt und den Inhalt dieses

Objekts mittels `var_dump()` ausgibt. Auf die Verwendung eines Templates habe ich in diesem Fall bewusst verzichtet, um das Beispiel kurz zu halten.

Beispiel

```
1 <?php
2
3 require_once 'src/Entities/Tag.php';
4
5 $tag = new Entities\Tag();
6 var_dump($tag);
```

index_test.php (Version 1)

Codebeispiel 11 *index_test.php (Version 1)*

Wenn Sie eine Klasse instanziieren möchten, die einen Namespace verwendet, so können Sie diesen einfach wie in Zeile 5 vor dem Namen der Klasse angeben.

Beispiel

```
1 <?php
2
3 require_once 'src/Entities/Tag.php';
4
5 use Entities\Tag;
6
7 $tag = new Tag();
8 var_dump($tag);
```

index_test.php (Version 2)

Codebeispiel 12 *index_test.php (Version 2)*

Wenn man in einem Controller eine Klasse häufiger nutzt oder nutzen will, so ist es sinnvoll, mit dem Schlüsselwort `use` einen Kurznamen zu aktivieren. Dies geschieht in Zeile 5 des Beispiels.

> Einen Kurznamen können Sie sowohl für den kompletten Namespace (inklusive Klassennamen) aktivieren als auch für einen Teil eines Namespace (ohne Angabe der Klasse). Der Kurzname entspricht immer der Angabe hinter dem letzten Backslash.

4.3 Konfiguration per Annotationen

Wenn bei einem Data Mapper die Datenhaltung von der Datenverwaltung getrennt ist, wo liegen dann bei Doctrine die Informationen, welche Tabellen welchen Datenklassen und welche Tabellenspalten welchen Attributen entsprechen? Für die Festlegung dieses sogenannten Datenbank-Schemas gibt es vier Möglichkeiten:

- Die unschönste Variante ist meiner Meinung nach jene, die auf reinem PHP-Code basiert.
- Die klassische Variante über *YAML* war bei Doctrine 1 der Standard. YAML ist eine vereinfachte Auszeichnungssprache (engl. markup language) zur Datenserialisierung.
- Die modernere Variante verwendet XML. Die Extensible Markup Language ist eine erweiterbare Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten. Die Varianten mit YAML und XML sind sehr ähnlich, sie verwenden lediglich eine unterschiedliche Syntax.
- Die neueste Variante verwendet Annotationen direkt in den Datenklassen und benötigt somit keine extra Dateien. Hierbei handelt es sich eigentlich um einfache Kommentare, die jedoch bestimmte Konventionen einhalten müssen. Als Annotation bezeichnet man jeden Kommentar-Begriff, der mit einem `@`-Zeichen beginnt. Diese Form des Kommentars wird auch als *DocComment* bezeichnet und kann von PHP direkt über die Reflection-API ausgelesen werden.

Mir gefällt die Variante mit den Annotationen am besten. Allerdings ist auch anzumerken, dass sie einen Nachteil hat: Die Datenklassen enthalten hiermit nämlich Informationen zum Datenbank-Schema, was eigentlich unserer Definition des Data Mapper-Entwurfsmusters widerspricht. Trotzdem scheinen sich Annotationen langsam als Quasistandard durchzusetzen.

Wir werden über Annotationen in der Datenklasse `Tag` Doctrine mitteilen, dass es sich hierbei um eine Datenbank-Entity handelt, wie die Tabelle in der Datenbank benannt ist und welche Datentypen die einzelnen Spalten haben. Da es sich danach nicht mehr um eine reine Datenklasse handelt, werden wir

solche Klassen fortan auch als Doctrine-Entities oder kurz ***Entities*** bezeichnen.

4.3.1 Entity

Um eine Datenklasse als ***Doctrine-Entity*** zu markieren, fügen wir unsere erste Annotation `@Doctrine\ORM\Mapping\Entity` direkt über der Klasse ein. Diese Annotation entspricht der Doctrine-Klasse `Doctrine\ORM\Mapping\Entity` im Ordner *vendor/doctrine/orm/lib*.

Beispiel

```php
<?php

namespace Entities;

/**
 * @Doctrine\ORM\Mapping\Entity
 */
class Tag
{
    // gekuerztes Beispiel
}
```

src/Entities/Tag.php (Version 3)

Codebeispiel 13 *src/Entities/Tag.php (Version 3)*

Das genügt bereits, um Doctrine diese Klasse als Entity bekannt zu machen.

4.3.2 Table

Den Namenskonventionen[6] entsprechend wird Doctrine eine Datenbank-Tabelle für die Entity erzeugen, die theoretisch exakt so heißt wie die Klasse selbst (nur ohne Namespace). Praktisch findet zumindest unter Windows abhängig von der MySQL-Konfiguration meist noch eine Umwandlung in Kleinbuchstaben statt. Wir erhalten also eine Tabelle namens `Tag` (oder `tag`), was wiederum unseren Vorgaben für die Beispiel-Datenbank zuwider läuft. Hier sollte die Tabelle laut physischem Datenmodell nämlich `tags` (Plural und zwingend Kleinbuchstaben) heißen.

---

6 Die Namenskonventionen sind in der Datei

vendor\doctrine\orm\lib\Doctrine\ORM\Mapping\DefaultNamingStrategy.php hinterlegt und können durch eigene Konventionen ersetzt werden (http://doctrine-orm.readthedocs.io/en/latest/reference/namingstrategy.html).

Beispiel

```php
<?php

namespace Entities;

/**
 * @Doctrine\ORM\Mapping\Entity
 * @Doctrine\ORM\Mapping\Table(name="tags")
 */
class Tag
{
    // gekuerztes Beispiel
}
```

src/Entities/Tag.php (Version 3b)

Codebeispiel 14 *src/Entities/Tag.php (Version 3b)*

Diese Änderung erreichen wir mit der Annotation `@Doctrine\ORM\Mapping\Table`, der wir als Parameter den gewünschten Tabellennamen übergeben können. Die Annotation entspricht der Doctrine-Klasse `Doctrine\ORM\Mapping\Table` im Ordner *vendor/doctrine/orm/lib*. Unsere Datenklasse enthält nun also Annotationen für zwei Doctrine-Klassen, die beide den Namespace `Doctrine\ORM\Mapping` nutzen.

Beispiel

```php
<?php

namespace Entities;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="tags")
 */
class Tag
{
    // gekuerztes Beispiel
}
```

src/Entities/Tag.php (Version 3c)
Codebeispiel 15 *src/Entities/Tag.php (Version 3c)*

Um Schreibarbeit bei diesem Namespace zu sparen, hat es sich eingebürgert, hierfür den Alias `ORM` zu definieren. Man verwendet dazu fast die gleiche Schreibweise wie bei der Aktivierung eines Kurznamens. Allerdings ergänzt man am Ende das Schlüsselwort `as` und dahinter den gewünschten Alias.

4.4 Der Primärschlüssel

Jede Entity benötigt Informationen über den eigenen Primärschlüssel (PK). Zu diesem Zweck müssen wir dem Attribut `$id`, das ja unser PK werden soll, drei Annotationen hinzufügen.

4.4.1 Id

Da wäre zum einen `@ORM\Id`, womit wir einfach mitteilen: Dieses Attribut hier ist der PK. Doctrine geht nämlich nicht davon aus, dass der PK immer ID heißen muss.

4.4.2 GeneratedValue

Durch `@ORM\GeneratedValue` teilen wir Doctrine mit, dass der Wert dieser Spalte automatisch generiert werden soll. Für MySQL heißt das `AUTO_INCREMENT`.

4.4.3 Column

Die Annotation `@ORM\Column` werden wir später noch oft benötigen. Damit legen wir fest, welchen Datentyp ein Attribut in der Datenbank haben soll. Bei einem synthetischen Primärschlüssel wie der ID ist dieser Typ natürlich `integer`.

Beispiel

```php
<?php

namespace Entities;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="tags")
 */
class Tag
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    protected $id = 0;

    // gekuerztes Beispiel
}
```

src/Entities/Tag.php (Version 4)

Codebeispiel 16 *src/Entities/Tag.php (Version 4)*

4.5 Doctrine-Datentypen

Neben `integer` kennt Doctrine weitere Datentypen, die einer eigenen Namensgebung folgen. Diese entsprechen nicht immer exakt einem Datentyp in MySQL (oder PHP), da Doctrine nicht nur MySQL als DBMS unterstützt. Hier eine Übersicht der wichtigsten Datentypen:

Doctrine	MySQL	PHP
`string`	`VARCHAR(255)` (Standard)	`string`
`text`	`LONGTEXT` (Standard)	`string`
`integer`	`INT/INTEGER`	`integer`
`decimal`	`DECIMAL(10,0)`	

	(Standard)	`string`[7]
`boolean`	`TINYINT(1)`	`boolean`
`date`	`DATE`	`DateTime`-Objekt
`time`	`TIME`	`DateTime`-Objekt
`datetime`	`DATETIME/TIMESTAMP`	`DateTime`-Objekt

7 Der Datentyp `double` wird nicht verwendet, um Rundungsprobleme zu vermeiden.

Tabelle 4.1 *Doctrine-Datentypen und ihre Zuordnung*

4.6 Parameter von Column

Der Annotation `@ORM\Column` können Sie (abhängig vom Datentyp) verschiedene Parameter übergeben.

4.6.1 type

Den Parameter `type` haben wir bereits eingesetzt. Er legt fest, welchen Datentyp unser Attribut haben soll.

4.6.2 length

Mit `length` legen Sie bei einem `string` oder `text` fest, wie viele Zeichen das Attribut enthalten darf und deaktivieren so den jeweiligen Doctrine-Standard. Die Annotation `@ORM\Column(type="string", length=50)` wird in der Datenbank beispielsweise zu `VARCHAR(50)` und ein `text` kann abhängig von `length` in der Datenbank zu `TINYTEXT`, `TEXT`, `MEDIUMTEXT` oder halt dem

Standard `LONGTEXT` werden.

4.6.3 unique

Mit `unique=true` legen Sie einen `UNIQUE`-Constraint auf die Tabellenspalte (auch Alternativschlüssel genannt). Es dürfen hier also keine zwei Datensätze den gleichen Wert aufweisen. Der Doctrine-Standard ist `unique=false`.

Beispiel

```php
<?php

namespace Entities;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="tags")
 */
class Tag
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    protected $id = 0;

    /**
     * @ORM\Column(type="string", length=25, unique=true)
     */
    protected $title = '';

    // gekuerztes Beispiel
}
```

src/Entities/Tag.php (Version 5)

Codebeispiel 17 *src/Entities/Tag.php (Version 5)*

Den Namenskonventionen entsprechend wird Doctrine eine Spalte für das Attribut `$title` erzeugen, die exakt so heißt wie das Attribut selbst (nur ohne `$`), d.h. Groß-/Kleinschreibung wird berücksichtigt. Wir erhalten hier also wie gewünscht eine Spalte namens `title`.

4.6.4 nullable

Im Gegensatz zum MySQL-Standard, bei dem alle Spalten `NULL` erlauben, legt Doctrine Spalten standardmäßig mit `NOT NULL` an, was eine gute Sache ist. Wenn Sie trotzdem bei einer Spalte `NULL` als Wert erlauben wollen, verwenden Sie `nullable=true`.

4.6.5 precision und scale

Diese Parameter werden für den Datentyp `decimal` benötigt und legen die Genauigkeit und die Anzahl der Nachkommastellen fest, sofern diese vom Doctrine-Standard abweichen sollen. Die Annotation `@ORM\Column(type="decimal", precision=6, scale=2)` wird in der Datenbank also zu `DECIMAL(6, 2)`. Bei zwei Nachkommastellen und einer Genauigkeit von sechs sind somit maximal vier Stellen vor dem Komma möglich.

> Mit Ausnahme der beiden Annotationen `@ORM\Entity` und `@ORM\Table` beziehen sich alle vorgestellten Annotationen auf ein bestimmtes Attribut einer Klasse. Wenn Sie somit ein Attribut definieren (siehe »Band 1: Grundlagen der OOP«), so erfolgt direkt über dieser Definition die Angabe der Annotationen für dieses Attribut.

4.7 Konfiguration des Autoloaders

Damit Sie die Entities unseres Beispiel-Projekts tatsächlich nutzen können, müssten Sie theoretisch für jede Klasse eine Einbindung mittels `require_once` vornehmen. Dies wäre jedoch viel zu viel Schreibarbeit, weswegen wir stattdessen das Autoloading von Composer auch für unsere eigenen Klassen nutzen. Hierfür teilen wir Composer in der *composer.json* mit, wo denn die entsprechenden Klassen zu finden sind. Dies geschieht im Abschnitt `autoload` für jeden Namespace (z. B. `Controllers` oder `Entities`) separat.

Beispiel

```
{
    "autoload": {
        "psr-0": {
            "Controllers": "./src/",
            "Entities": "./src/",
            "Repositories": "./src/"
        }
    },
    "require": {
        "php": ">=5.4",
        "doctrine/orm": "2.5.5"
    }
}
```

composer.json

Codebeispiel 18 *composer.json*

Wir legen alle unsere per Autoloading eingebundenen Klassen in Unterordnern von *src* ab. Dies dient dazu, diese Klassen besser von manuell zu ladenden Dateien unterscheiden zu können. Der Name des Unterordners (z. B. `Entities`) entspricht hierbei immer dem Namespace der Klasse, weswegen der Autoloader problemlos den nötigen Dateipfad auf Basis dieses Namespaces bestimmen kann (z. B. *src/Entities*). Der Klassenname `Tag` wird dann nur noch mit der Dateiendung `.php` versehen, und schon kennt der Autoloader die komplette Pfadangabe für die Datei (z. B. *src/Entities/Tag.php*) in unserem Projekt.

Der Namespace `Repositories` kommt übrigens erst bei den fortgeschrittenen Techniken ganz am Ende dieses Bandes zum Einsatz.

Übung 5:

1. Ergänzen Sie im *src*-Ordner einen Ordner *Validators*.
2. Ergänzen Sie das noch fehlende Autoloading für diesen neuen Ordner in der Datei *composer.json*. Führen Sie anschließend ein `php composer.phar install` durch.

Den Ordner *src/Validators* benötigen wir beim Validieren. Die hierbei verwendeten Validator-Klassen sind jedoch **nicht** Doctrine-Standard. Ich habe

deswegen im Doctrine-Skeleton auf diesen Unterordner verzichtet. Rufen Sie das Konsolenkommando `php composer.phar install` auf, nachdem Sie diesen Ordner angelegt und im Autoloading ergänzt haben, so erscheint nachfolgende Ausgabe:

```
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
Warning: The lock file is not up to date with the latest change
You may be getting outdated dependencies. Run update to update
Nothing to install or update
Generating autoload files
```

Composer-Ausgabe - install

Codebeispiel 19 *Composer-Ausgabe - install*

Wie Sie der Ausgabe entnehmen können, wurde zwar das Autoloading wie gewünscht aktualisiert (siehe letzte Zeile), doch Composer war nicht wirklich glücklich und warnte uns vor einem potenziellen Problem. Dies liegt daran, dass bei einer Änderung in der *composer.json* eine neue Abhängigkeit oder eine neue Versionsangabe vermutet wird. Dies ist aktuell jedoch nicht der Fall, weswegen Sie die Meldung ignorieren können.

Allerdings will ich kurz auf ein weiteres Composer-Kommando vorgreifen, da dieses in der Warnung erwähnt wird. Es handelt sich um `php composer.phar update`.

```
Loading composer repositories with package information
Updating dependencies (including require-dev)
Nothing to install or update
Writing lock file
Generating autoload files
```

Composer-Ausgabe - update

Codebeispiel 20 *Composer-Ausgabe - update*

Auch hier würde wie gewünscht das Autoloading aktualisiert. Allerdings würde dabei der Inhalt der *composer.lock* ignoriert und es würde nach neuen Abhängigkeiten bzw. neueren Versionen unserer Abhängigkeiten gesucht.

> Nach jedem Hinzufügen von Abhängigkeiten in der *composer.json* ist eine Aktualisierung mit dem Konsolenkommando `php composer.phar update`

nötig. Da hierbei jedoch auch aktuellere Versionen bereits vorhandener Abhängigkeiten installiert würden, verwende ich immer eindeutige Versionsnummern.[8]

---

8 Dies könnte man jedoch auch umgehen, indem man nur die Pakete eines bestimmten Anbieters oder sogar nur ein einzelnes Paket aktualisiert (https://getcomposer.org/doc/03-cli.md#update).

4.8 Zusammenfassung

In dieser Lektion haben Sie gelernt:

- wie Sie mittels Namespaces Namenskollisionen zwischen eigenem und fremdem Code vermeiden.
- wie Sie dem Autoloading von Composer zusätzliche Namespaces beibringen.
- wie Sie eine Aktualisierung mit Composer vornehmen.

Außerdem haben Sie die Datenklasse für die Datenbank-Tabelle `tags` implementiert. Sie haben gelernt, wie Sie aus einer einfachen Datenklasse mit Hilfe von Annotationen eine Doctrine-Entity machen.

Sie sind nun in der Lage, nahezu beliebige isolierte Tabellen (zu Datenbank-Beziehungen kommen wir später) als Doctrine-Entities zu implementieren. Letztlich ändern sich bei jeder Klasse ja nur der Tabellenname (und somit der Klassenname und die Annotationen der Klasse), die Spaltennamen (und damit die Attribute, Getter und Setter) sowie die Annotationen der einzelnen Attribute.

4.9 Testen Sie Ihr Wissen

1. Nennen Sie Beispiele für Namespaces jenseits der PHP-Programmierung.
2. Wir verwenden Annotationen zur Festlegung des Datenbank-Schemas. Was ist der primäre Unterschied zu den drei anderen Varianten?

3. Überlegen Sie, ob jedes Attribut Annotationen (z. B. `@ORM\Column`) benötigt.

4.10 Aufgaben zur Selbstkontrolle

Übung 6:

Implementieren Sie die in dieser Lektion vorgestellte Datenklasse für die Tabelle `tags` in der Datei *src/Entities/Tag.php* oder kopieren Sie sich diese aus dem Begleitmaterial.

Übung 7:

Machen Sie aus dieser Datenklasse mit Hilfe von Annotationen eine Doctrine-Entity. Vergessen Sie hierbei nicht den Namespace, den Alias und die Annotationen der einzelnen Attribute. Wir behalten hierbei den Doctrine-Standard `nullable=false` bei. Da es sich hierbei um den Standard handelt, können Sie die Angabe weglassen.

5 Aufbau einer Datenbank-Verbindung

In dieser Lektion lernen Sie
• wie Sie mit Doctrine eine Datenbank-Verbindung aufbauen. • wozu der EntityManager dient.

5.1Einleitung

Sie haben nun zwar eine Doctrine-Entity vorbereitet, können entsprechende `Tag`-Objekte aber noch nicht mit Doctrine in der Datenbank ablegen. Dies liegt daran, dass wir noch nirgends unsere Zugangsdaten hinterlegt haben.

5.2 Bootstrapping

Die Zugangsdaten werden in unserem Fall in der Datei *default-config.php* im Ordner *config* hinterlegt. Die Datei ist Teil der bereits vorhandenen Projekt-Struktur, enthält jedoch noch nicht die korrekten Angaben.

Beispiel

```php
<?php

// MySQL database configuration
$connectionOptions = array(
    'driver' => 'pdo_mysql',
    'host' => 'localhost',
    'user' => 'root',
    'password' => '',
    'dbname' => 'newsticker_d2',
);

// Application/Doctrine configuration
$applicationOptions = array(
```

```
14      'debug_mode' => true, // in production environment false
15      'entity_dir' => dirname(__DIR__) . '/src/Entities',
16  );
```

config/default-config.php

Codebeispiel 21 *config/default-config.php*

Die Verbindungsdaten `$connectionOptions` in den Zeilen 4-10 sollte jeder in ähnlicher Form kennen, der schon einmal mit PHP und MySQL gearbeitet hat. Hier legen Sie fest, wie und mit welchen Zugangsdaten sich Doctrine mit der Datenbank verbinden soll.

Oh ja, eine Datenbank brauchen wir natürlich auch noch! In unserem Newsticker gehe ich von einer MySQL-Datenbank auf `localhost` mit dem Namen `newsticker_d2` aus. Als Benutzername und Passwort werden `root` und ein leeres Passwort angenommen. Passen Sie diese Angaben bitte an Ihre Gegebenheiten (z. B. kein leeres Passwort) an.

Übung 8:

1. Erstellen Sie die (noch) leere Datenbank `newsticker_d2`. Achten Sie hierbei auf eine korrekte utf-8-Kollation (z. B. `utf8_general_ci`).
2. Rufen Sie den Front-Controller *index.php* des Newstickers im Browser auf. Achten Sie hierbei darauf, dass die URL das Projekt-Verzeichnis *newsticker_d2* beinhalten muss.[9] Was passiert?

   9 Die URL sollte die Form `http://meinserver/newsticker_d2/index.php` haben. Auf einem lokal installierten Webserver entspricht `meinserver` normalerweise `localhost` (ohne Portangabe).

3. Geben Sie nun die korrekten Verbindungsdaten für die neue Datenbank in der *config/default-config.php* an.

Wie Sie beim Testen bemerkt haben sollten, sind Doctrine die hinterlegten Zugangsdaten zunächst einmal total egal. Diese kommen erst zum Tragen, wenn wir in der nächsten Lektion tatsächlich auf die Datenbank zugreifen.

5.2.1 $applicationOptions

In den Zeilen 13-16 wird das Konfigurations-Array `$applicationOptions` mit Daten befüllt. Dieses Array betrifft im Gegensatz zu den Verbindungsdaten in `$connectionOptions` Ihre Anwendung selbst. Interessant ist hierbei vor allem Zeile 14, mit der wir den sogenannten Debug-Mode aktivieren. Dieser erleichtert uns das Debugging unserer Anwendung während des Programmierens.

Das Array `$applicationOptions` enthält jedoch noch einen zweiten Wert, der die eine oder andere Frage aufwerfen dürfte. PHP stellt jedem Skript zur Laufzeit eine Reihe von vordefinierten Konstanten zur Verfügung. Zu diesen sogenannten magischen Konstanten gehört `__DIR__`. Mit ihr bringen Sie in Erfahrung, in welchem Pfad sich die Datei befindet, in der wir die Konstante verwenden (hier *default-config.php*). Wir wissen natürlich, dass sich die Datei im Ordner *config* befindet. Doch es ist vom jeweiligen Entwicklungsrechner (z. B. Betriebssystem) abhängig, wo sich dieser Ordner wiederum befindet. Aus einer übergebenen Zeichenkette, die den Pfad zu einer Datei oder einem Verzeichnis enthält, gibt nun die Funktion `dirname()` den Pfad des übergeordneten Verzeichnisses zurück. D.h. wir erhalten so den Pfad zum Hauptordner unseres Newsticker-Projekts und verketten diesen mit dem String '/src/Entities', um den (absoluten) Pfad zu unserem Ordner *Entities* zu erhalten.

5.2.2 EntityManager

Doch wo wird die Konfigurationsdatei überhaupt eingebunden? Befassen wir uns zur Klärung dieser Frage ein wenig mit der durch Composer installierten Projekt-Struktur.

Beispiel

```
1 <?php
2
3 // Load config file
4 require_once __DIR__ . '/../config/default-config.php';
5
6 // Use Composer autoloading
7 require_once __DIR__ . '/../vendor/autoload.php';
8
9 // Get Doctrine entity manager
```

```
10 use Doctrine\ORM\Tools\Setup;
11 use Doctrine\ORM\EntityManager;
12
13 $proxyDir = null;
14 $cache = null;
15 $isSimpleMode = false;
16
17 $config = Setup::createAnnotationMetadataConfiguration(
18     array($applicationOptions['entity_dir']),
19     $applicationOptions['debug_mode'],
20     $proxyDir,
21     $cache,
22     $isSimpleMode
23 );
24
25 $em = EntityManager::create($connectionOptions, $config);
```

inc/bootstrap.inc.php

Codebeispiel 22 *inc/bootstrap.inc.php*

Die Einbindung geschieht in der Datei *bootstrap.inc.php*, welche sich im Ordner *inc* Ihrer Projekt-Struktur befindet. Sie bereitet die Benutzung von Doctrine vor, d.h. sie aktiviert beispielsweise in Zeile 7 das Autoloading. Den gesamten Vorgang der Vorbereitung bezeichnet man auch als Bootstrapping.

Im Rahmen dieses Bootstrappings werden zur besseren Lesbarkeit unseres Codes in Zeile 10 und 11 zwei Kurznamen für die benötigten Klassen aktiviert. Abschließend wird dann in beiden Klassen jeweils eine statische Methode aufgerufen. Die Rückgabewerte dieser Methoden werden dann in den Variablen `$config` und `$em` aufgefangen. In der ersten Variable befindet sich ein Objekt, welches auf Basis unserer beiden Konfigurations-Arrays und der drei Variablen `$proxyDir`, `$cache` und `$isSimpleMode` erstellt wurde. Bezüglich der drei Variablen sei nur so viel gesagt, dass hiermit Doctrine-Features deaktiviert bzw. durch den fehlenden Inhalt nicht aktiviert werden. Der EntityManager in `$em` ist jedoch wesentlich interessanter. Er stellt nämlich unsere Schnittstelle zu Doctrine dar. Ob wir Datensätze auslesen, neue speichern oder alte löschen, alles passiert über dieses Objekt. Es ist eine Namenskonvention, die Variable, in der das `EntityManager`-Objekt liegt, als `$em` zu benennen. Der EntityManager ist das zentrale Objekt, über das sämtliche Doctrine-Operationen ablaufen. Sie werden also wirklich noch sehr oft mit ihm zu tun haben.

Beispiel

```php
<?php

require_once 'inc/functions.inc.php';
require_once 'inc/helper.inc.php';

require_once 'inc/bootstrap.inc.php';

// Session needed for flash messages
session_start();

// Path to our index.php
$basePath = dirname(__FILE__);

$controller = isset($_GET['controller']) ? $_GET['controller
$action = isset($_GET['action']) ? $_GET['action'] : 'index'

$controllerNamespace = 'Controllers\\';
$controllerName = $controllerNamespace . ucfirst($controller

if (class_exists($controllerName)) {
    $requestController = new $controllerName($basePath, $em)
    $requestController->run($action);
} else {
    $requestController = new Controllers\IndexController($ba
    $requestController->render404();
}
```

index.php

Codebeispiel 23 *index.php*

Das Bootstrapping wird wiederum von unserem Front-Controller *index.php* in Zeile 6 gestartet. Wenn Sie sich nun den Front-Controller noch etwas genauer ansehen, dann sollten Ihnen gegenüber »Band 1: Grundlagen der OOP« ein paar Neuerungen auffallen. Diese finden Sie in den Zeilen 9, 17-18, 21 und 24. In Zeile 9 wird zunächst einmal eine Session gestartet. Diese benötigen wir später für die Methoden `AbstractBase#setMessage()` und `AbstractBase#getMessage()`. In den Zeilen 17, 18 und 24 berücksichtigen wir, dass unsere Controller-Klassen nun den Namespace `Controllers` verwenden. Eine vergleichbare Aufgabe übernimmt später die Methode `AbstractBase#getControllerShortName()`, welche beim automatischen Ermitteln des Templates den Namespace wieder entfernt. In den Zeilen 21 und 24 übergeben wir dann dem Konstruktor der Controller-Klasse den in Zeile 12 ermittelten Pfad der *index.php* und den EntityManager. Beides wird dann in der Controller-Instanz in den Attributen `$basePath` und `$em` abgelegt.

5.3 Testen Sie Ihr Wissen

1. Wie nennt man die Schnittstelle zu Doctrine, die man zum Auslesen, Speichern und Löschen von Datensätzen nutzt?

2. Es gibt eine Namenskonvention für die Variable, in der das Objekt dieser Schnittstelle liegt. Wie lautet diese?

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.

6.1Einleitung

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

```
1 <?php
2
```

```
 3 require_once 'inc/bootstrap.inc.php';
 4
 5 $schemaTool = new \Doctrine\ORM\Tools\SchemaTool($em);
 6
 7 $factory = $em->getMetadataFactory();
 8 $metadata = $factory->getAllMetadata();
 9
10 try {
11     $schemaTool->updateSchema($metadata);
12 } catch (PDOException $e) {
13     echo 'ACHTUNG: Bei der Aktualisierung des Schemas gab es
14     echo $e->getMessage() . "<br />";
15     if (preg_match("/Unknown database '(.*)'/", $e->getMessa
16         die(
17             sprintf(
18                 'Erstellen Sie die Datenbank %s mit der Kolla
19                 $matches[1]
20             )
21         );
22     }
23 }
24
25 ?>
26 Das Schema-Tool wurde durchlaufen.
```

setup.php

Codebeispiel 24 *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 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 hierbei 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 Projekt-Struktur enthält deswegen bereits eine noch ziemlich leere Controller-Klasse `IndexController`.

Beispiel

```php
<?php

namespace Controllers;

class IndexController extends AbstractBase
{
    public function indexAction()
    {
        // Fill me
    }
}
```

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

Codebeispiel 25 *src/Controllers/IndexController.php (Version 1 - Controller-Skeleton)*

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             array('title' => 'PHP')
13         );
14
15         $this->addContext('tag', $tag);
16     }
17 }
```

src/Controllers/IndexController.php (Version 2)

Codebeispiel 26 *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

```
<p>
    Es wurde das Tag <strong><?php echo $tag; ?></strong>
    mit der ID <?php echo $tag->getId(); ?> angelegt.
</p>
```

templates/IndexController/indexAction.tpl.php

Codebeispiel 27 *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.

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);
```

IndexController.php - Erweiterung der Standard-Aktion

Codebeispiel 28 *IndexController.php - Erweiterung der Standard-Aktion*

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

```php
<?php

namespace Controllers;

use Entities\Tag;

class IndexController extends AbstractBase
{
    public function indexAction()
    {
        $tag = new Tag(
            array('title' => 'PHP' . time())
        );

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

        $this->addContext('tag', $tag);
    }
}
```

src/Controllers/IndexController.php (Version 3)

Codebeispiel 29 *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 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 Aufgaben zur Selbstkontrolle

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 Aufgaben

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;')`.

Hinweis: Die Methode `Connection#query()` ermöglicht das Absetzen von nativen SQL-Anweisungen. Mit der Anweisung `TRUNCATE TABLE` können Sie eine Tabelle in einen jungfräulichen Zustand zurückversetzen. Alle Datensätze werden gelöscht und der Autoinkrement-Wert wird zurückgesetzt.

7 Datenbankabfragen mit Doctrine

In dieser Lektion lernen Sie
• wie Sie das EntityRepository einer Tabelle erhalten. • wie Sie mit diesem Repository alle oder einzelne Datensätze auslesen.

7.1Einleitung

Eine MySQL-Datenbank mit `SELECT` zu befragen, sollte für Sie kein Problem sein. Das Gleiche in PHP mit Hilfe von Doctrine umzusetzen, ist ebenfalls schnell erledigt.

7.2 EntityRepository

Ein `EntityRepository` repräsentiert eine Tabelle mit allen enthaltenen Datensätzen und dient dazu, Datensätze nach bestimmten Kriterien auszulesen.

Mit der Methode `EntityManager#getRepository()` erhalten Sie ein solches ***Repository*** (dt. Ablage bzw. Behältnis) für eine bestimmte Entity, deren Namen Sie inklusive Namespace als Parameter angeben müssen.

Beispiel

```
$em = $this->getEntityManager();
$em->getRepository('Entities\Tag');
```

Repository holen

Codebeispiel 30 *Repository holen*

Dieses Repository ist eine Instanz der Klasse `Doctrine\ORM\EntityRepository`.

7.2.1 Chaining

Die Repository-Instanz selbst wird selten in einer Variablen abgelegt, da man sie lediglich für den Aufruf der eigentlichen Methoden benötigt. Stattdessen wird an den ersten Methodenaufruf mit dem Pfeiloperator eine weitere Methode angekettet und nur deren Rückgabewert in einer Variablen abgelegt. Eine solche Verkettung von zwei Methodenaufrufen (engl. ***chaining***) ist immer dann möglich, wenn der erste Methodenaufruf ein Objekt als Rückgabewert liefert. Hierbei ist ein Chaining natürlich nicht nur auf zwei Methodenaufrufe begrenzt, wichtig ist lediglich, dass die vorhergehenden Aufrufe immer den korrekten Datentyp zurückliefern. Allerdings sollte man es auch nicht übertreiben.

7.2.2 findAll

Mit `EntityRepository#findAll()` erhalten Sie alle gespeicherten Datensätze der entsprechenden Tabelle, wobei sie als ein Array von Objekten dieser Klasse zurückkommen.

Beispiel

```
$tags = $em
    ->getRepository('Entities\Tag')
    ->findAll()
;
```

Chaining - Repository holen ->findAll()

Codebeispiel 31 *Chaining - Repository holen ->findAll()*

> Die Methode `EntityRepository#findAll()` gibt ein leeres Array zurück, wenn sie keine Datensätze findet.

7.2.3 find

Ebenso häufig benötigen wir aber einen bestimmten Datensatz als Objekt, den wir anhand seiner ID identifizieren. Die Methode `EntityRepository#find()` arbeitet im Prinzip wie `EntityRepository#findAll()`, nur übergeben wir als

Parameter dieses Mal die ID (z. B. die ID 1) des gewünschten Datensatzes und erhalten als Rückgabewert direkt das Objekt.

Beispiel

```
$tag = $em
    ->getRepository('Entities\Tag')
    ->find(1)
;
```

Chaining - Repository holen ->find()

Codebeispiel 32 *Chaining - Repository holen ->find()*

> Die Methode `EntityRepository#find()` gibt `null` zurück, wenn sie keinen Datensatz passend zur ID findet.

7.2.4 findBy

Doctrine stellt uns für jedes Attribut eine eigene Finder-Methode zur Verfügung. Die Namenskonvention lautet hier: Der Name beginnt mit `findBy`, gefolgt von dem Namen des Attributs. Das Ganze wird selbstverständlich in ***lowerCamelCase***-Schreibweise notiert. Für das Attribut `$title` wäre der Finder also `findByTitle()`.

Dieser Methode übergeben wir den Wert, nach welchem wir suchen wollen, und Doctrine führt dann eine SQL-Abfrage in der Art von `SELECT * FROM tags WHERE title = ?;` aus. Als Rückgabewert erhalten wir ein Array von Objekten. Dies gilt auch, wenn nur ein Datensatz gefunden wird.

Beispiel

```
$tags = $em
    ->getRepository('Entities\Tag')
    ->findByTitle('PHP')
;
```

Chaining - Repository holen ->findBy

Codebeispiel 33 *Chaining - Repository holen ->findBy*

Die Methode `EntityRepository#findBy()` gibt ein leeres Array zurück, wenn sie keine passenden Datensätze findet.

7.2.5 findOneBy

Analog zu der Gruppe von `findBy`-Methoden gibt es `findOneBy`, wo nur maximal ein Datensatz zurückgeliefert wird. Diese Finder können Sie verwenden, wenn Sie ohnehin wissen, dass nur ein Objekt zurückgegeben werden kann. Dies ist beispielsweise bei Attributen mit `UNIQUE`-Constraint der Fall.

Beispiel

```
$tag = $em
    ->getRepository('Entities\Tag')
    ->findOneByTitle('PHP')
;
```

Chaining - Repository holen ->findOneBy

Codebeispiel 34 *Chaining - Repository holen ->findOneBy*

Die Methode `EntityRepository#findOneBy()` gibt `null` zurück, wenn sie keinen passenden Datensatz findet.

7.3 Zusammenfassung

Sie sind nun in der Lage, mittels der beschriebenen Repository-Methoden eines oder mehrere Objekte aus der Datenbank auszulesen. Sofern eine Methode mehrere Datensätze zurückliefert, so geschieht dies immer als Array von Objekten.

Diese Repository-Variante ist die einfachste Form, `SELECT`-Anweisungen mit Doctrine auszuführen, aber auch die unflexibelste. In der nächsten Lektion werden Sie mächtigere, aber auch komplexere Methoden des Zugriffs kennenlernen.

7.4 Testen Sie Ihr Wissen

1. Was repräsentiert bei Doctrine eine Tabelle mit allen enthaltenen Datensätzen und dient gleichzeitig dazu, Datensätze nach bestimmten Kriterien auszulesen?

2. Man legt das EntityRepository selten in einer eigenen Variablen ab, sondern hängt einen weiteren Methodenaufruf an den ersten an. Wie nennt man diese Vorgehensweise?

7.5 Aufgaben zur Selbstkontrolle

Übung 15:

In Lektion 6 haben Sie die Methode `IndexController#indexAction()` und das zugehörige Template *indexAction.tpl.php* mit Inhalt versehen. Erstellen Sie nun in einer neuen Controller-Klasse `TagController` die Methode `addAction` und benutzen Sie dort eine Methode des Repositories, um **alle** Tags zu ermitteln. Geben Sie dann im Template *templates/TagController/editAction.tpl.php* den `title` jedes Tags in einer Schleife aus.

Um direkt der einen oder anderen Frage vorzubeugen. Ja, das Template soll **nicht** *addAction.tpl.php* benannt sein. Sie müssen also die Methode `AbstractBase#setTemplate()` aufrufen, um das Standard-Template *addAction.tpl.php* für die Aktion **nicht** zu verwenden! Es soll auch wirklich nur der Titel ausgegeben werden, ein simpler `var_dump` reicht hier **nicht** aus.

Übung 16:

Rufen Sie nun diese neue Methode im Browser auf. Denken Sie hierbei daran, dass die URL die nachfolgende Form haben sollte:

```
http://meinserver/newsticker_d2/index.php?controller=tag&action=add
```

Übung 17:

Optimieren Sie die Ausgabe im Template *editAction.tpl.php*. Geben Sie hierzu die Tags als Liste aus. Die Ausgabe aller Tags soll also in einem `ul`-Tag und die Ausgabe jedes einzelnen Titels in einem separaten `li`-Tag erfolgen.

8 Komplexe Abfragen mit Doctrine

In dieser Lektion lernen Sie
• was DQL ist und wie es sich von SQL unterscheidet. • wie Sie mit DQL Datensätze auslesen. • welche Vorteile der QueryBuilder von Doctrine bietet.

8.1Einleitung

Sehr viele Fälle, in denen Sie Objekte aus der Datenbank holen möchten, können Sie schon mit den Repository-Methoden erschlagen. Für den Fall, dass die Abfragen komplexer werden, stellt uns Doctrine gleich zwei mächtige Alternativen zur Verfügung. Welche Sie verwenden, bleibt Ihnen überlassen, ich bevorzuge den `QueryBuilder`.

8.2 Per DQL

Doch wenden wir uns zuerst ***DQL*** zu. Der Begriff DQL (***Doctrine Query Language***) klingt nicht umsonst fast wie SQL. Es handelt sich im Prinzip um SQL, das von Doctrine um einige Konzepte erweitert wurde. Der Hauptunterschied besteht darin, dass Sie mit DQL nicht Tabellen befragen, sondern die Entities. Wir sagen also nicht mehr `SELECT ... FROM tags`, sondern `SELECT ... FROM Tag`, oder genauer `SELECT ... FROM Entities\Tag`, da wir den Namespace angeben müssen, wenn die Entity-Klasse in einem Namespace liegt.

Es ist außerdem erforderlich, der Klasse einen kurzen Aliasnamen zu geben, da Sie den Namen sehr oft referenzieren müssen. Ansonsten ist DQL syntaktisch weitgehend identisch zu SQL, wir haben aber den Vorteil, dass wir nun Objekte bei unseren Abfragen erhalten.

Beispiel

```
SELECT t FROM Entities\Tag t WHERE t.id = 1
```

Der Buchstabe `t` ist in diesem Beispiel der Aliasname. Die Definition des Alias erfolgt direkt hinter der Angabe der Entity. Er muss vor allen Attributen angegeben werden (z. B. bei `t.id`) und dient zudem als Ersatz für den `*` direkt hinter dem `SELECT` im normalen SQL. Die vollständige Dokumentation zu DQL finden Sie unter http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html.

8.2.1 createQuery

Ein DQL-Query erzeugen wir mit der Methode `EntityManager#createQuery()`, der wir das DQL als String übergeben.

Beispiel

```
$query = $em->createQuery(
    "SELECT t FROM Entities\Tag t WHERE t.id > 2"
);
$query = $em->createQuery(
    "SELECT t FROM Entities\Tag t WHERE t.id BETWEEN 2 AND 5"
);
$query = $em->createQuery(
    "SELECT t FROM Entities\Tag t WHERE t.title = 'PHP'"
);
$query = $em->createQuery(
    "SELECT t FROM Entities\Tag t WHERE t.title LIKE '%Doctrine
);
```

Wir erhalten in `$query` dann jeweils ein Objekt der Klasse `Doctrine\ORM\Query`.

8.2.2 getResult

Dieses `Query`-Objekt repräsentiert nun unser SQL-Statement. Mit der Methode `Query#getResult()` holen wir uns das Ergebnis als Array von Objekten. Beachten Sie, dass das Query auch erst in dem Moment ausgeführt wird, in

dem diese Methode aufgerufen wird.

Beispiel

```
$query = $em->createQuery(
    "SELECT t FROM Entities\Tag t WHERE t.title = 'PHP'"
);
$tags = $query->getResult();
```

Per DQL

Codebeispiel 35 *Per DQL*

> Wenn Sie die Meldung `Doctrine\ORM\Query\QueryException: [Syntax Error] (...) Expected Literal, got '"'` oder `Doctrine\ORM\Query\QueryException: [Syntax Error] (...) Expected Doctrine\ORM\Query\Lexer::T_STRING, got '"'` erhalten, so haben Sie in einer `WHERE`-Bedingung doppelte anstatt einfacher Anführungszeichen um einen Wert vom Datentyp String benutzt.

Prepared Statements

Jeder PHP-Programmierer kennt das Problem: Es ist, vorsichtig ausgedrückt, eine Qual, SQL-Anweisungen mit PHP-Variablen zu modifizieren. Auch Sie haben sicherlich schon Code wie diesen geschrieben:

Beispiel

```
$sql = 'SELECT * FROM tags WHERE title = "' . $title . '"';
```

SQL mit String-Verknüpfung

Codebeispiel 36 *SQL mit String-Verknüpfung*

Die Variable `$title` steht für eine Benutzereingabe. Zum Sicherheitsaspekt solcher Benutzereingaben kommen wir übrigens im Rahmen von Lektion 14. Jetzt soll es jedoch erstmal um das Problem von String-Verknüpfungen in SQL gehen. Gerade bei Variablen, die Strings enthalten, hat man in solchen Fällen besonders viel Spaß mit den unterschiedlichen Anführungszeichen. Dabei ist

dieses Statement, wenn wir ehrlich sind, noch äußerst simpel.

Daher möchte ich Ihnen eine der besten Erfindungen der Datenbank-Programmierung vorstellen: ***Prepared Statements***. Der ein oder andere Leser mag diese schon von ***PDO*** kennen. Sie unterscheiden sich von normalen Datenbank-Abfragen dadurch, dass das Definieren des SQLs bzw. DQLs und die Ausführung in mindestens zwei Schritte aufgetrennt werden. Jetzt werden Sie sich vielleicht wundern, denn diese Trennung haben wir bereits mit `EntityManager#createQuery()` und `Query#getResult()` behandelt, und in Bezug auf String-Verknüpfungen hat sie uns bis jetzt keinen Vorteil gebracht.

Der Vorteil von ***Prepared Statements*** liegt jedoch darin, dass in ihnen (benannte) ***Platzhalter*** verwendet werden können, die Sie in einem zusätzlichen Schritt mit Werten befüllen.

Beispiel

```
$dql = 'SELECT t FROM Entities\Tag t';
$dql .= ' WHERE t.title = :title';

// Schritt 1: Vorbereiten
$query = $em->createQuery($dql);

// Schritt 2: Befuellen
$query->setParameter('title', 'PHP');

// Schritt 3: Ausfuehren
$tags = $query->getResult();
```

DQL und benannte Platzhalter

Codebeispiel 37 *DQL und benannte Platzhalter*

Benannte Platzhalter verwenden die Syntax `:platzhalter`, also z. B. `:id` oder `:title`. Erst der Doppelpunkt vor dem Namen sagt aus, dass es sich um einen benannten Platzhalter handelt. Sie dürfen ihn nicht weglassen.

Der Methode `Query#setParameter()` übergeben wir nun zwei Parameter, wobei der erste Parameter dem Namen des Platzhalters entspricht, nur ohne den Doppelpunkt und als String notiert. Aus :title wird also `'title'`.

> Beachten Sie, dass um die Platzhalter im DQL keine Anführungszeichen stehen dürfen, da diese bei Strings automatisch hinzugefügt werden.

Falls der gleiche Platzhalter im Query mehrfach vorkommen sollte, so müssen Sie den Wert der Methode `Query#setParameter()` nur einmal übergeben. Ansonsten gilt: Sie müssen `Query#setParameter()` exakt so oft aufrufen, wie Sie unterschiedlich benannte Platzhalter verwendet haben.

Beispiel

```
$dql = 'SELECT t FROM Entities\Tag t';
$dql .= ' WHERE t.id = :id AND t.title = :title';

// Schritt 1: Vorbereiten
$query = $em->createQuery($dql);

// Schritt 2: Befuellen mittels assoziativem Array
$data = array('id' => 1, 'title' => 'PHP');
$query->setParameters($data);

// Schritt 3: Ausfuehren
$tags = $query->getResult();
```

DQL und benannte Platzhalter

***Codebeispiel 38** DQL und benannte Platzhalter*

Wenn Sie mehrere Platzhalter in einem Query benötigen, so bietet sich alternativ eine Befüllung mit der Methode `Query#setParameters()` an. Als Parameter muss ein assoziatives Array übergeben werden, dessen Schlüssel allen Platzhaltern (ohne Doppelpunkt) entsprechen.

Limitierungen von Platzhaltern in Prepared Statements

Sie können Platzhalter nur für Werte verwenden, niemals als Ersatz für einen Tabellen- oder Spaltennamen in SQL bzw. als Ersatz für Entities oder Attribute in DQL. Folgende Varianten sind also beispielsweise **nicht erlaubt**:

- `SELECT t FROM :entity t WHERE t.id = 1`
- `SELECT t FROM Entities\Tag t WHERE t.:attribut = 1`

8.2.3 getSingleResult

Die Methode `Query#getSingleResult()` tut prinzipiell das Gleiche wie `Query#getResult()`, nur liefert sie lediglich einen Ergebnis-Datensatz als Objekt zurück. Diese Methode ist für Fälle gedacht, wo Sie lediglich einen Datensatz erwarten und auch immer erhalten.

> Liefert das Query keinen oder mehrere Datensätze, so führt dies bei der Ausführung der Methode `Query#getSingleResult()` zu einer `NoResultException` oder einer `NonUniqueResultException`.

8.2.4 getOneOrNullResult

Die Methode `Query#getOneOrNullResult()` unterscheidet sich lediglich in einem Detail von der Methode `Query#getSingleResult()`. Sie liefert nämlich `null` zurück, sofern sie keinen Datensatz findet.

> Liefert das Query mehrere Datensätze, so führt dies bei der Ausführung der Methode `Query#getOneOrNullResult()` zu einer `NonUniqueResultException`.

Beispiel

```
$query = $em->createQuery(
    "SELECT t FROM Entities\Tag t WHERE t.id > 1"
);
$tags = $query->getOneOrNullResult();
```

NonUniqueResultException

Codebeispiel 39 *NonUniqueResultException*

Da wir drei Datensätze in der Tabelle `tags` haben und davon lediglich einer die ID 1 hat, würde diese DQL-Abfrage mit einer `NonUniqueResultException` scheitern. Doch dies lässt sich im DQL durch eine Beschränkung der Ergebnisdatensätze mit `LIMIT` leicht beheben.

Beispiel

```
$query = $em->createQuery(
    "SELECT t FROM Entities\Tag t WHERE t.id > 1 LIMIT 1"
);
$tags = $query->getOneOrNullResult();
```

NonUniqueResultException gelöst

Codebeispiel 40 *NonUniqueResultException gelöst*

Allerdings wäre erst durch eine zusätzliche Sortierung mit `ORDER BY` genau festgelegt, welchen der möglichen beiden Datensätze wir als Ergebnis erhalten. So würde man beispielsweise mit `ORDER BY t.id LIMIT 1` denjenigen Datensatz erhalten, der die kleinste ID hat. Mit `ORDER BY t.id DESC LIMIT 1` würde man hingegen den Datensatz erhalten, der die größte ID hat.

8.3 Per QueryBuilder

Der ***QueryBuilder*** ist ein Aufsatz auf DQL und bietet ein Methoden-Interface für das Erzeugen von DQL. Anstatt DQL als einen String zu schreiben (und sich dort zu vertippen), rufen wir für jeden Teilbereich eine Methode auf, die so heißt wie das DQL-Gegenstück.

Anstatt `SELECT t from Entities\Tag t WHERE t.title = 'PHP'` schreiben wir also `select('t')->from('Entities\Tag', 't')->where("t.title = 'PHP'")`.

Der Vorteil des `QueryBuilder` ist, dass Sie durch die einzelnen Methoden im Fehlerfall eher Feedback erhalten, wo Sie sich vertippt oder eine Pflichtangabe (Parameter) vergessen haben. Außerdem finde ich die Syntax mit angeketteten Methoden übersichtlicher. Hier gilt aber: YMMV[10].

---

10 *Your mileage may vary* (https://en.wiktionary.org/wiki/your_mileage_may_vary).

8.3.1 createQueryBuilder

Mit der Methode `EntityManager#createQueryBuilder()` erzeugen Sie eine neue Instanz der Klasse `Doctrine\ORM\QueryBuilder`. An dieses Objekt können

Sie dann die eigentlichen Methoden des QueryBuilders *anketten* und schließlich mit der Methode `EntityManager#getQuery()` daraus ein normales Query-Objekt erzeugen.

Beispiel

```
$query = $em
    ->createQueryBuilder()
    ->select('t')
    ->from('Entities\Tag', 't')
    ->where('t.title = :title')
    ->setParameter('title', 'PHP')
    ->getQuery()
;
$tags = $query->getResult();
```

Per QueryBuilder

Codebeispiel 41 *Per QueryBuilder*

8.3.2 Die wichtigsten Methoden

Sehen wir uns nun einen Überblick über die wichtigsten Methoden des QueryBuilder an.

- `QueryBuilder#select()`: Diese Methode repräsentiert das `SELECT`-Statement im DQL und wird hier eigentlich nur zum Selektieren des Klassen-Alias (oder der Aliasse) verwendet.
- `QueryBuilder#from()`: Diese Methode steht für das DQL-`FROM` und benötigt zwei Parameter, den Klassennamen und einen Alias, der ebenfalls ein Pflichtfeld darstellt.
- `QueryBuilder#where()`: Entspricht der ersten `WHERE`-Bedingung im DQL.
- `QueryBuilder#andWhere()`: Eine zweite (dritte ...) `WHERE`-Bedingung, mit `AND` mit dem Vorgänger verknüpft.
- `QueryBuilder#orWhere()`: Eine zweite (dritte ...) `WHERE`-Bedingung, mit `OR` mit dem Vorgänger verknüpft.
- `QueryBuilder#orderBy()`: Steht für das `ORDER BY` im DQL und akzeptiert zwei Parameter, das Attribut, nach dem sortiert wird, und die Richtung,

also `ASC` oder `DESC`.

- `QueryBuilder#setParameter()`: Mit dieser Methode können Sie einem benannten Parameter im DQL einen Wert zuweisen.
- `QueryBuilder#setParameters()`: Mit dieser Methode können Sie mehreren benannten Parametern im DQL ihre Werte als assoziatives Array zuweisen.
- `QueryBuilder#setMaxResults`: Entspricht dem `LIMIT` im DQL.
- `QueryBuilder#setFirstResult`: Entspricht dem `OFFSET` im DQL.
- `QueryBuilder#getQuery()`: Wandelt den QueryBuilder in ein Objekt der Klasse `Doctrine\ORM\Query` um, das Sie dann weiterverarbeiten können.

Beispiel

```
$query = $em
    ->createQueryBuilder()
    ->select('t')
    ->from('Entities\Tag', 't')
    ->orderBy('t.id', 'DESC')
    ->getQuery()
;
$tags = $query->getResult();
```

Sortierung mit orderBy

Codebeispiel 42 *Sortierung mit orderBy*

Die vollständige Dokumentation zum QB finden Sie unter http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html.

8.4 Fluent Interfaces

Wie Sie im vorherigen Kapitel eventuell bemerkt haben, verwenden wir beim Chaining mit Ausnahme von `EntityManager#createQueryBuilder()` nur Methoden der Klasse `QueryBuilder`. Doch wie ist es möglich, dass wir mehrere Methoden der gleichen Klasse verketten können? Zur Erklärung möchte ich einen kleinen Exkurs am Beispiel einer Klasse `Date` machen.

Beispiel

```php
<?php

class Date
{
    protected $year;
    protected $month;
    protected $day;

    public function __construct($year, $month, $day)
    {
        $this->setYear($year);
        $this->setMonth($month);
        $this->setDay($day);
    }

    public function __toString()
    {
        return sprintf(
            '%04d-%02d-%02d',
            $this->year,
            $this->month,
            $this->day
        );
    }

    public function setYear($year)
    {
        $this->year = $year;
    }

    public function setMonth($month)
    {
        $this->month = $month;
    }

    public function setDay($day)
    {
        $this->day = $day;
    }
}
```

Date.php (Version 1)

Codebeispiel 43 *Date.php (Version 1)*

Diese Klasse ist wirklich simpel. Wir haben Setter für unsere drei Attribute und einen Konstruktor, der diese bei der Instanziierung aufruft. Außerdem gibt es noch die magische Methode `__toString()`, damit wir die Objekte dieser Klasse

mit `echo` ausgeben können. Eine Instanziierung und Befüllung der Attribute würde beispielsweise folgendermaßen aussehen:

Beispiel

```
$date1 = new Date(1970, 1, 1);
```

Instanziierung und Befüllung (Version 1)

Codebeispiel 44 *Instanziierung und Befüllung (Version 1)*

Schon an diesem Beispiel sollte ersichtlich sein, dass es ein kleines Problem mit der Lesbarkeit des Codes gibt. Ohne uns die Definition der Methode anzusehen, können wir nämlich nicht sagen, ob der zweite Parameter der Monat oder der Tag ist.

Nehmen wir nun an, dass es eine neue Anforderung für die Klasse gibt. Alle drei Parameter sollen nun optional sein und stattdessen soll bei Bedarf der aktuelle Zeitpunkt verwendet werden.

Beispiel

```
<?php

class Date
{
    protected $year;
    protected $month;
    protected $day;

    public function __construct($year = null, $month = null,
    {
        $year = !empty($year) ? $year : date('Y');
        $this->setYear($year);

        $month = !empty($month) ? $month : date('m');
        $this->setMonth($month);

        $day = !empty($day) ? $day : date('d');
        $this->setDay($day);
    }

    public function __toString()
    {
```

```
23          return sprintf(
24              '%04d-%02d-%02d',
25              $this->year,
26              $this->month,
27              $this->day
28          );
29      }
30
31      public function setYear($year)
32      {
33          $this->year = $year;
34      }
35
36      public function setMonth($month)
37      {
38          $this->month = $month;
39      }
40
41      public function setDay($day)
42      {
43          $this->day = $day;
44      }
45  }
```

Date.php (Version 2)

Codebeispiel 45 *Date.php (Version 2)*

Der Konstruktor ist nun schon etwas komplexer und es offenbart sich ein weiteres Problem, wenn wir die optionalen Angaben weglassen wollen.

Beispiel

```
$date2 = new Date(1970, 1); // ohne Tag
$date3 = new Date(1970, null, 1); // ohne Monat
$date4 = new Date(null, 1, 1); // ohne Jahr
```

Instanziierung und Befüllung (Version 2)

Codebeispiel 46 *Instanziierung und Befüllung (Version 2)*

Wollen wir den Wert für den ersten oder zweiten Parameter nicht übergeben, so geht dies nicht so einfach. Wir können diese nämlich nicht einfach weglassen, sondern müssen stattdessen einen leeren Wert angeben (z. B. `null`).

Ich würde den Code jedoch gerne so verbessern, dass die Befüllung beliebiger

Parameter optional und der Code trotzdem lesbar ist.

Beispiel

```php
<?php

class Date
{
    protected $year;
    protected $month;
    protected $day;

    public function __construct(array $params = array())
    {
        $year = !empty($params['year']) ? $params['year'] : 
        $this->setYear($year);

        $month = !empty($params['month']) ? $params['month']
        $this->setMonth($month);

        $day = !empty($params['day']) ? $params['day'] : dat
        $this->setDay($day);
    }

    public function __toString()
    {
        return sprintf(
            '%04d-%02d-%02d',
            $this->year,
            $this->month,
            $this->day
        );
    }

    public function setYear($year)
    {
        $this->year = $year;
    }

    public function setMonth($month)
    {
        $this->month = $month;
    }

    public function setDay($day)
    {
        $this->day = $day;
    }
```

```
45 }
```

Date.php (Version 3)

Codebeispiel 47 *Date.php (Version 3)*

Der Konstruktor ist noch etwas komplexer geworden. Doch wie sieht es mit dem zur Benutzung nötigen Code aus?

Beispiel

```
$date1 = new Date(array('year' => 1970, 'month' => 1, 'day' =>
$date2 = new Date(array('year' => 1970, 'month' => 1)); // ohne
$date3 = new Date(array('year' => 1970, 'day' => 1)); // ohne M
$date4 = new Date(array('month' => 1, 'day' => 1)); // ohne Jah
```

Instanziierung und Befüllung (Version 3a)

Codebeispiel 48 *Instanziierung und Befüllung (Version 3a)*

Die Länge des zur Befüllung benötigten Codes ist zwar gestiegen, durch die assoziativen Schlüssel wissen wir nun jedoch genau, was welcher Wert bedeutet. Außerdem können wir beliebige Schlüssel im Array weglassen, d.h. diese sind genauso optional wie zuvor die drei einzelnen Parameter. Richtig gut lesbar ist unser Code jedoch immer noch nicht, da man beispielsweise sehr schnell eine der beiden schließenden runden Klammern vergisst.

Beispiel

```
$date = new Date(
    array(
        'year' => 1970,
        'month' => 1,
        'day' => 1,
    )
);
```

Instanziierung und Befüllung (Version 3b)

Codebeispiel 49 *Instanziierung und Befüllung (Version 3b)*

Dies könnte man zwar durch manuelle Zeilenumbrüche lösen, doch ich habe einen besseren Vorschlag, nämlich die sogenannten ***Fluent Interfaces***.

Der Begriff ***Fluent Interfaces*** wurde 2005 von *Eric Evans* und *Martin Fowler*

geprägt. Frühe Implementierungen finden sich jedoch schon in den 70er und 80er Jahren. Da gab es halt nur noch keinen griffigen Namen für das Konzept.

Beispiel

```php
<?php

class Date
{
    protected $year;
    protected $month;
    protected $day;

    public function __construct()
    {
        $this->setYear(date('Y'));
        $this->setMonth(date('m'));
        $this->setDay(date('d'));
    }

    public function __toString()
    {
        return sprintf(
            '%04d-%02d-%02d',
            $this->year,
            $this->month,
            $this->day
        );
    }

    public function setYear($year)
    {
        $this->year = $year;

        return $this;
    }

    public function setMonth($month)
    {
        $this->month = $month;

        return $this;
    }

    public function setDay($day)
    {
        $this->day = $day;

```

```
44          return $this;
45      }
46  }
```

Date.php (Version 4)

Codebeispiel 50 *Date.php (Version 4)*

Der Konstruktor ist nun wieder sehr einfach und gut lesbar, da hier lediglich die drei Setter aufgerufen werden und diesen der aktuelle Zeitpunkt übergeben wird. Hiermit erhält also jedes Attribut seinen Standardwert, welcher dann mit den Settern verändert werden kann.

Beispiel

```
$date = new Date();
$date
    ->setYear(1970)
    ->setMonth(1)
    ->setDay(1)
;
```

Instanziierung und Befüllung (Version 4)

Codebeispiel 51 *Instanziierung und Befüllung (Version 4)*

Wir können nun ein beliebiges Chaining der drei Methoden vornehmen. Es gibt weder Vorgaben, in welcher Reihenfolge wir dies oder ob wir es überhaupt tun müssen. Erreicht wird dies dadurch, dass jeder unserer drei Setter als Rückgabewert `$this` benutzt. Nach einem Setter-Aufruf haben wir also immer ein Objekt, welches wir für ein Chaining mit dem nächsten Methodenaufruf verwenden können.

Genau dieses Konzept der ***Fluent Interfaces*** benutzt der `QueryBuilder` von Doctrine. Es ist also egal, ob wir zuerst `QueryBuilder#select()` oder `QueryBuilder#from()` aufrufen. Wir können auch zuerst eine `WHERE`-Bedingung mit `QueryBuilder#where()` angeben. Im Gegensatz zu dem `Date`-Beispiel sind jedoch nicht alle Methoden optional und auch die Reihenfolge ist nicht ganz beliebig.

Beispiel

```
$query = $em
    ->createQueryBuilder() // Pflicht, 1. Angabe
    ->select('t') // Pflicht
    ->from('Entities\Tag', 't') // Pflicht
    ->orderBy('t.id', 'DESC') // optional
    ->getQuery() // Pflicht, letzte Angabe
;
$tags = $query->getResult();
```

Chaining-Vorgaben QueryBuilder

***Codebeispiel 52** Chaining-Vorgaben QueryBuilder*

Der Aufruf von `EntityManager#createQueryBuilder()` muss grundsätzlich zuerst erfolgen, da wir erst durch diesen überhaupt ein Objekt erhalten, welches den Zugriff auf die restlichen Methoden ermöglicht. Der Aufruf von `QueryBuilder#getQuery()` muss als letzte Angabe des Chainings erfolgen, da wir erst hierdurch das `Query`-Objekt erhalten, welches wir beispielsweise für den Aufruf von `Query#getResult()` benötigen.

8.5 Zusammenfassung

Sie haben in dieser Lektion zwei weitere Varianten von `SELECT`-Anweisungen kennengelernt. Mit diesen neuen Varianten sind Sie nun auch in der Lage, komplexere Datenbank-Abfragen zu formulieren. Bei der Verwendung des QueryBuilders hilft uns ein ***Fluent Interface***.

8.6 Testen Sie Ihr Wissen

1. Wie nennt sich das bei Doctrine verwendete Äquivalent zu SQL und was ist der wichtigste Unterschied zu SQL?

2. Was verwendet man als Rückgabewert der Methoden, um in einer Klasse ein einfaches Fluent Interface umzusetzen?

8.7 Aufgaben zur Selbstkontrolle

Übung 18:

Erstellen Sie im Controller `IndexController` eine neue Methode `searchAction`. Diese Methode soll mittels ***Prepared Statement*** alle Tags auslesen, deren ID zwischen 1 und 3 liegt. Verwenden Sie für die beiden Integers Platzhalter. Aktuell können Sie das Template *templates/TagController/editAction.tpl.php* bei der Ausgabe der Datensätze wiederverwerten, was mit dem optionalen zweiten Parameter der Methode `AbstractBase#setTemplate()` möglich ist. Rufen Sie den Front-Controller im Browser auf und geben Sie hierbei die Aktion `search` an. Sie sollten nun drei Tags angezeigt bekommen.

Übung 19:

Ändern Sie den Inhalt der Aktion so ab, dass Sie alle Tags auslesen, die den Buchstaben `h` enthalten. Auch hierbei soll ein Platzhalter zum Einsatz kommen. Sortieren Sie die Ausgabe alphabetisch nach dem Titel. Prüfen Sie auch die Browser-Ausgabe der geänderten Aktion.

9 Die Webmasters Doctrine Extensions

In dieser Lektion lernen Sie
• welche Vorteile die Webmasters Doctrine Extensions bieten. • was Sie beim Einsatz dieses Pakets beachten sollten.

9.1Einleitung

Um Ihnen den weiteren Umgang mit Doctrine etwas zu erleichtern, habe ich bereits ein wenig Schreibarbeit für Sie erledigt. Diese Arbeit stelle ich Ihnen in Form der *Webmasters Doctrine Extensions* auf *Packagist* zur Verfügung.

9.2 Doctrine Extensions

Doch dies ist nicht das einzige Paket, welches wir jetzt installieren werden. Für Lektion 10 benötigen wir nämlich auch noch die sogenannten *Gedmo Doctrine Extensions*.

Beispiel

```
{
    "autoload": {
        "psr-0": {
            "Controllers": "./src/",
            "Entities": "./src/",
            "Repositories": "./src/",
            "Validators": "./src/"
        }
    },
    "require": {
        "php": ">=5.4",
        "doctrine/orm": "2.5.5",
        "gedmo/doctrine-extensions": "2.4.24",
```

```
14            "webmasters/doctrine-extensions": "4.0.0"
15        }
16    }
```

composer.json

Codebeispiel 53 *composer.json*

Zur Installation dieser beiden Erweiterungen können Sie die *composer.json* wie dargestellt ergänzen und danach das Konsolenkommando `php composer.phar update` verwenden. Alternativ geht die Installation eines Packagist-Pakets aber auch nur auf der Konsole:

```
php composer.phar require gedmo/doctrine-extensions:2.4.24
```

Mit dem `require`-Kommando wird in einem Rutsch die *composer.json* ergänzt und das entsprechende Paket installiert. Existiert noch gar keine *composer.json*, so wird die Datei zunächst für uns angelegt und dann die Paket-Installation vorgenommen.

Man kann so entweder jedes Paket einzeln installieren oder mehrere Pakete in einem Rutsch:

```
php composer.phar require gedmo/doctrine-extensions:2.4.24 webr
```

Übung 20:

Installieren Sie die beiden Doctrine-Erweiterungen mit *Composer*.

Nach der Installation finden Sie die Klassen der beiden Pakete in den Ordnern *vendor/gedmo/doctrine-extensions/lib* und *vendor/webmasters/doctrine-extensions/lib*.

9.3 Bootstrapping

Wenn man Doctrine mit den *Gedmo Doctrine Extensions* nutzen möchte, benötigt man ungefähr 100 Zeilen Code für das ***Bootstrapping***. Da dieser Code sehr fehleranfällig ist, möchte ich Ihnen diese Tortur ersparen. Doch

sehen wir uns zunächst unseren bisherigen Code an:

Beispiel

```php
<?php

// Load config file
require_once __DIR__ . '/../config/default-config.php';

// Use Composer autoloading
require_once __DIR__ . '/../vendor/autoload.php';

// Get Doctrine entity manager
use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

$proxyDir = null;
$cache = null;
$isSimpleMode = false;

$config = Setup::createAnnotationMetadataConfiguration(
    array($applicationOptions['entity_dir']),
    $applicationOptions['debug_mode'],
    $proxyDir,
    $cache,
    $isSimpleMode
);

$em = EntityManager::create($connectionOptions, $config);
```

inc/bootstrap.inc.php (Version 1)

Codebeispiel 54 *inc/bootstrap.inc.php (Version 1)*

Die Datei *bootstrap.inc.php* enthält nur wenige wirklich wichtige Zeilen Code, die für jede Controller-Klasse unserer Anwendung benötigt werden, nämlich die Einbindung der Konfigurationsdatei, die Aktivierung des Autoloadings und die Befüllung von `$em`. Es spricht also nichts dagegen, dass wir den unnötigen Ballast einer Bibliothek überlassen.

Beispiel

```php
<?php

// Load config file
require_once __DIR__ . '/../config/default-config.php';

```

```php
// Use Composer autoloading
require_once __DIR__ . '/../vendor/autoload.php';

// Get Doctrine entity manager
use Webmasters\Doctrine\Bootstrap;

$bootstrap = Bootstrap::getInstance(
    $connectionOptions,
    $applicationOptions
);

$em = $bootstrap->getEm();
```

inc/bootstrap.inc.php (Version 2)

Codebeispiel 55 *inc/bootstrap.inc.php (Version 2)*

Der neue Code beginnt in Zeile 10. Hier legen wir zunächst einen Kurznamen an, damit Zeile 12 kürzer und somit besser lesbar wird. Der statischen Methode `Bootstrap#getInstance()` übergeben wir dann unsere beiden Konfigurations-Arrays. Als Rückgabewert erhalten wir eine Instanz der Klasse `Webmasters\Doctrine\ORM\Bootstrap`. Diese können wir dann in Zeile 17 mit der Methode `Bootstrap#getEm()` nach einer Instanz des EntityManagers fragen.

Meiner Meinung nach ist der Code in der *config/default-config.php* und *inc/bootstrap.inc.php* immer noch etwas zu lang. Deswegen habe ich mich entschlossen, ein paar Standards in der `Bootstrap`-Klasse zu hinterlegen, d.h. Sie können durch Beachtung meiner Konventionen Code sparen.

Beispiel

```php
<?php

// MySQL database configuration
$connectionOptions = array(
    'driver' => 'pdo_mysql',
    'host' => 'localhost',
    'user' => 'root',
    'password' => '',
    'dbname' => 'newsticker_d2',
);
```

config/default-config.php (Version 1)

Codebeispiel 56 *config/default-config.php (Version 1)*

```php
<?php

// Use Composer autoloading
require_once __DIR__ . '/../vendor/autoload.php';

// Get Doctrine entity manager
use Webmasters\Doctrine\Bootstrap;

$bootstrap = Bootstrap::getInstance();
$em = $bootstrap->getEm();
```

inc/bootstrap.inc.php (Version 3)

Codebeispiel 57 *inc/bootstrap.inc.php (Version 3)*

Beim Bootstrapping wird nun standardmäßig davon ausgegangen, dass im Ordner *config* eine Konfigurationsdatei mit dem Array `$connectionOptions` existiert. Die Konfigurationsdatei kann entweder den Dateinamen *default-config.php* tragen oder einen Namen verwenden, der spezifisch für den aktuellen Webserver ist.

> Der Dateiname der alternativen Konfigurationsdatei ist abhängig vom Host-Namen des Webservers und wird mittels `php_uname('n') . '-config.php'` ermittelt. Sind mehrere Konfigurationsdateien vorhanden, so wird zunächst überprüft, ob es eine alternative Datei für den aktuellen Webserver gibt. Ist dies nicht der Fall, so werden die Konfigurationsdaten aus der *default-config.php* verwendet. Bei dieser Fallunterscheidung ist es übrigens unerheblich, ob die Verbindungsdaten auf dem aktuellen Webserver auch funktionieren, d.h. Sie sind selbst für korrekte Angaben verantwortlich.

9.3.1 Pflichtangaben

Die Angabe der `$connectionOptions` ist weiterhin verpflichtend, da meine Bibliothek Ihre Verbindungsdaten nicht erraten kann. Die Angabe des Arrays `$applicationOptions` ist nun jedoch komplett optional, d.h. bei einem Verzicht greifen unsere bisherigen Angaben als Standardwerte. Der Debug-Mode ist also beispielsweise aktiv.

Beispiel

```php
<?php

// MySQL database configuration
$connectionOptions = array(
    'driver' => 'pdo_mysql',
    'host' => 'localhost',
    'user' => 'root',
    'password' => '',
    'dbname' => 'newsticker_d2',
);

// Application/Doctrine configuration
$applicationOptions = array(
    'debug_mode' => false, // Production server
);
```

config/default-config.php (Version 2)

Codebeispiel 58 *config/default-config.php (Version 2)*

> Auf einem produktiven Webserver sollte man den Debug-Mode **immer** deaktivieren! Auf einem Entwicklungsrechner sollte man dies hingegen auf keinen Fall machen. Wie sollte man sonst die eigenen Programmier-Fehler finden?

9.3.2 Vererbung

Wenn man sich den Inhalt von `$em` mittels `var_dump()` ansieht, so sollte eine Sache sofort auffallen. Es handelt sich um eine Instanz der Klasse `Webmasters\Doctrine\ORM\EntityManager` und nicht der Original-Klasse `Doctrine\ORM\EntityManager`.

Beispiel

```php
<?php

namespace Webmasters\Doctrine\ORM;

class EntityManager extends \Doctrine\ORM\EntityManager
{
    // gekuerztes Beispiel
}
```

vendor/webmasters/doctrine-

extensions/lib/Webmasters/Doctrine/ORM/EntityManager.php

Codebeispiel 59 *vendor/webmasters/doctrine-extensions/lib/Webmasters/Doctrine/ORM/EntityManager.php*

Schauen Sie sich einmal den dargestellten Codeausschnitt der Klasse genauer an. Was fällt auf? Genau, die Klasse nutzt eine Vererbung von der Elternklasse `Doctrine\ORM\EntityManager`. Sie erbt also alle Attribute und Methoden dieser Elternklasse. Die beiden EntityManager-Klassen unterscheiden sich in der Verwendung zunächst einmal nicht. Es wird jedoch eine neue Methode `EntityManager#getValidator()` ergänzt, welche wir beim Thema ***Validierungen*** in Lektion 13 einsetzen werden.

Durch den **Backslash** vor dem Namespace `Doctrine\ORM` der Elternklasse drücken Sie übrigens aus, dass sich dieser nicht im aktuellen Namespace `Webmasters\Doctrine\ORM` der Kindklasse befindet.

9.4 ArrayMapper

Sehen Sie sich nun die Klasse *src/Entities/Tag.php* in Ihrem Projekt-Verzeichnis *newsticker_d2* an. Fällt Ihnen sofort ein Anwendungsfall ein, wo auch eine Vererbung von Nutzen sein könnte? Stellen Sie sich einmal vor, Sie erstellen die noch fehlenden Doctrine-Entities `User` und `Article`. Welche wichtige Methode ist bei allen drei Klassen identisch? Richtig, die Methode `setData()`. Eine solche Duplizierung von Code sollte man jedoch möglichst vermeiden.

> Eine Duplizierung von Code verstößt meist gegen das sogenannte DRY-Prinzip (engl. »Don't Repeat Yourself«, dt. »Wiederhole dich nicht«). Dieses Prinzip ist auch bekannt als »once and only once« (dt. »Einmal und nur einmal«). Vermeiden lässt sie sich normalerweise durch eine Auslagerung (z. B. Partials bei Templates und Funktionen/Methoden beim restlichen Code).

Allerdings lässt sich dieses Problem in unserem aktuellen Fall mittels Vererbung nur sehr unschön lösen, da man bei `extends` nur exakt eine Elternklasse angeben kann. Um uns diese Möglichkeit offenzuhalten, brauchen wir also eine

andere Lösung, den `ArrayMapper`.[11]

---

11 Alternativ könnte man auch die mit PHP 5.4 eingeführten Traits verwenden. Ich wollte jedoch bei meinem Paket keine höheren Voraussetzungen als Doctrine selbst haben. Siehe http://php.net/de/language.oop5.traits

Die Klasse `Webmasters\Doctrine\ORM\Util\ArrayMapper` bietet drei öffentliche Methoden.

- Die statische Methode `ArrayMapper#setEntity()` erhält als Parameter ein Objekt einer unserer Doctrine-Entities. Als Rückgabewert liefert sie ähnlich der Vorgehensweise von `Bootstrap#getInstance()` eine Instanz der eigenen Klasse `ArrayMapper`. Zum Einsatz kommt hierbei wie schon beim Thema ***Fluent Interfaces*** gezeigt `$this`.

- Diese Instanz ermöglicht Ihnen die Nutzung von zwei weiteren Methoden, nämlich von `ArrayMapper#setData()` und `ArrayMapper#toArray`. Die erste Methode ist der Ersatz für die uns schon bekannte Methode `setData()` (bzw. `setDaten()`) und dient dazu, auf Basis eines assoziativen Arrays die Setter einer Klasse aufzurufen. Die zweite Methode ist der invertierte Anwendungsfall: Sie ruft die Getter einer Klasse auf und liefert deren Werte als assoziatives Array. Beachten Sie jedoch bitte, dass die Methode `ArrayMapper#toArray` im Gegensatz zu `__toString()` keine ***magische Methode*** ist.

> Damit die Methoden `ArrayMapper#setData()` und `ArrayMapper#toArray()` funktionieren, müssen in den Doctrine-Entities für jedes Attribut ein Getter und ein Setter vorhanden sein. Lediglich der Setter für die ID bildet eine Ausnahme und wir verzichten bewusst auf ihn.

Doch wie müssen wir unsere Doctrine-Entity `Tag` anpassen, um den `ArrayMapper` zu verwenden?

Beispiel

```
1 <?php
2
```

```php
namespace Entities;

use Doctrine\ORM\Mapping as ORM;
use Webmasters\Doctrine\ORM\Util;

/**
 * @ORM\Entity
 * @ORM\Table(name="tags")
 */
class Tag
{
    // gekuerztes Beispiel

    public function __construct(array $data = array())
    {
        Util\ArrayMapper::setEntity($this)->setData($data);
    }

    // gekuerztes Beispiel
}
```

src/Entities/Tag.php

Codebeispiel 60 *src/Entities/Tag.php*

Um die Lesbarkeit des Codes zu verbessern, wird in Zeile 6 ein Kurzname für den Namespace `Webmasters\Doctrine\ORM\Util` festgelegt. Danach ersetzen wir im Konstruktor `$this->setData($data)` mit einem Aufruf des `ArrayMapper`. Hierbei nutzen wir ein Chaining von zwei Methoden, da wir die Instanz von `ArrayMapper` lediglich für den Aufruf der zweiten Methode benötigen. Die Methode `setData()` können wir nach dieser Änderung aus unserer Doctrine-Entity löschen.

9.5 Zusammenfassung

In dieser Lektion haben Sie gelernt:

- welche Parameter die `Bootstrap`-Klasse benötigt.
- wie Sie mit der Methode `Bootstrap#getEm()` eine Instanz des EntityManagers erhalten.
- wie man durch eine Nutzung der `ArrayMapper`-Klasse eine unnötige Code-

Duplizierung vermeidet.

9.6 Testen Sie Ihr Wissen

1. Welches Konfigurations-Array ist durch das überarbeitete Bootstrapping nun optional?
2. Welches Paket haben wir neben den *Webmasters Doctrine Extensions* in dieser Lektion noch installiert?

9.7 Aufgaben zur Selbstkontrolle

Übung 21:

Implementieren Sie den Endstand der *bootstrap.inc.php* im Unterordner *inc*. Passen Sie auch Ihre Konfigurationsdatei an.

Übung 22:

Ändern Sie den Konstruktor der `Tag`-Klasse so, dass er den `ArrayMapper` nutzt. Vergessen Sie nicht, die nun überflüssige Methode `setData()` zu löschen.

Übung 23:

Rufen Sie abschließend den Front-Controller auf. Wenn Sie alles richtig gemacht haben, sollten Sie keine Fehlermeldung angezeigt bekommen.

10 DateTime

In dieser Lektion lernen Sie
• wie Sie DateTime-Objekte in PHP verwenden und welche Vorteile diese bieten. • wie Sie eine nützliche Doctrine-Erweiterung nutzen. • welche Besonderheiten Sie beim Doctrine-Datentyp `datetime` berücksichtigen müssen.

10.1Einleitung

In der Programmierung arbeitet man ständig mit Datums- und Zeitwerten, doch lange Zeit lief man hier schnell in das eine oder andere Problem. Dies spiegelt sich auch in der Vielzahl von Funktionen zu diesem Thema wider. So könnten Ihnen beispielsweise die Funktionen `time()`, `mktime()`, `strtotime()`, `strftime()` oder `date()` durchaus bekannt vorkommen, um nur die wichtigsten Vertreter dieser Fraktion zu nennen.

Dieses Problem erhielt jedoch spätestens mit PHP 5.2.2 einen besseren Lösungsansatz in Form der (fehlerbereinigten) `DateTime`-Klasse. Die Nutzung von DateTime-Objekten für unsere Datums- und Zeitwerte bringt einige handfeste Vorteile mit sich, die ich im Folgenden vorstellen möchte.

10.2 Die DateTime-Klasse

Eine Instanz der `DateTime`-Klasse kann wie bei den meisten anderen Klassen auch mit dem Schlüsselwort `new` erzeugt werden.

Beispiel

```
$datetime1 = new DateTime('1970-01-01'); // angegebenes Datum
```

```php
$datetime2 = new DateTime('1970-01-01 13:29'); // Datum plus Uh
```

DateTime - Datum/Uhrzeit

Codebeispiel 61 *DateTime - Datum/Uhrzeit*

Wollen Sie ein DateTime-Objekt für einen bestimmten Zeitpunkt erzeugen, so nutzen Sie am besten die ***ISO 8601***-Schreibweise (**JJJJ-MM-TT**). Wird keine Uhrzeit angegeben, so wird `00:00:00.000000` benutzt.

> Wird dem Konstruktor kein gültiges Datum (z. B. `'1970-011-01'` oder `'abc'`) als Wert für den ersten Parameter übergeben, so erhält man die Meldung `Fatal error: Uncaught exception 'Exception' with message 'DateTime::__construct().`

Beispiel

```php
$datetime1 = new DateTime('now'); // aktuelles Datum plus Uhrze
$datetime2 = new DateTime();
$datetime3 = new DateTime(null);

$datetime4 = new DateTime('');
$datetime5 = new DateTime(false);
```

DateTime - now

Codebeispiel 62 *DateTime - now*

Soll für den aktuellen Zeitpunkt ein Objekt erzeugt werden, so geschieht dies mit dem String `'now'`, durch das Weglassen jeglicher Werte für Parameter oder durch Übergabe von `null` als Wert für den ersten Parameter.

> Wird dem Konstruktor ein leerer String (`''`) oder der Boolean `false` als Wert für den ersten Parameter übergeben, so wird ebenfalls der aktuelle Zeitpunkt verwendet. Für unseren weiteren Code ist dieses Verhalten (insbesondere bei einem leeren String) jedoch nicht wünschenswert. Dazu bald mehr.

Beispiel

```php
$datetime1 = new DateTime('+2 days'); // 'now' plus 2 Tage
$datetime2 = new DateTime('+1 week'); // 'now' plus 1 Woche
```

DateTime - Modifier von now

Codebeispiel 63 *DateTime - Modifier von now*

Wie Sie durch die Beispiele eventuell bemerkt haben, entsprechen die erlaubten Werte für diesen Konstruktor-Parameter weitestgehend dem ersten Parameter von `strtotime()`.

10.2.1 format

In Ordnung, wir haben nun ein DateTime-Objekt, doch wie geben wir dieses wieder aus?

Beispiel

```php
$datetime = new DateTime();
echo $datetime; // Fehler
```

DateTime - fehlerhafte Ausgabe

Codebeispiel 64 *DateTime - fehlerhafte Ausgabe*

Diese Variante funktioniert leider nicht; Sie würden die Meldung `Object of class DateTime could not be converted to string` erhalten. Erinnert Sie diese Fehlermeldung an etwas? Die fehlende `__toString()`-Methode ist ein bekanntes Problem und es sieht leider auch nicht so aus, als ob diese in nächster Zeit in die Klasse `DateTime` Einzug halten würde. Doch wie ist dann die derzeit korrekte Vorgehensweise?

Beispiel

```php
$datetime = new DateTime();

echo $datetime->format('Y-m-d'); // ISO 8601
echo $datetime->format('d.m.Y'); // deutsche Schreibweise
echo $datetime->format('H:i:s'); // Uhrzeit
```

DateTime - format

Codebeispiel 65 *DateTime - format*

Sie müssen die Methode `DateTime#format()` benutzen, um eine Ausgabe im gewünschten Format zu erreichen. Als Parameter müssen Sie einen Format-String angeben, der die gleichen Möglichkeiten wie der erste Parameter von `date()` bietet.

Doch wenn wir Ähnliches mit `strtotime()` und `date()` erreichen könnten, wo liegen dann die Vorteile der DateTime-Klasse?

10.2.2 modify

Beispiel

```php
$datetime = new DateTime('1970-01-01');

$datetime->modify('+1 week'); // 1 Woche danach
$datetime->modify('-2 days'); // 2 Tage davor

var_dump($datetime); // insgesamt 5 Tage danach
```

DateTime - modify

Codebeispiel 66 *DateTime - modify*

Oftmals benötigt man für Vergleiche einen Zeitpunkt, der um eine bestimmte Anzahl von Wochen oder Tagen vom ursprünglichen Wert abweicht. Dies ist mit `DateTime#modify()` glücklicherweise ohne die leidige Rechnung **60 Sekunden * 60 Minuten * 24 Stunden (* 7 Tage)** möglich.

> Die Anwendung von `DateTime#modify()` ist destruktiv, d. h. der vorherige Zeitpunkt ist nicht mehr im Objekt enthalten.

10.2.3 diff

Beispiel

```php
$datetime1 = new DateTime('1970-01-01');
```

```php
$datetime2 = new DateTime('now');

$differenz = $datetime2->diff($datetime1);
echo $differenz->format('%a'); // (absolute) Tage
```

DateTime - diff

***Codebeispiel 67** DateTime - diff*

In diesem Beispiel ermitteln wir, wie viele Tage seit dem Beginn der Unix-Zeitzählung schon vergangen sind. Wichtig ist, dass `DateTime#diff()` eine Instanz der Klasse `DateInterval` zurückgibt, die ebenfalls über keine `__toString()`-Methode verfügt. Auch hier ist deswegen ein `DateInterval#format()` und ein Format-String als Parameter nötig. Der String `%a` ist natürlich nicht die einzige Möglichkeit, jedoch werden wir in dieser Lektion nur diese Formatierung benötigen.

10.3 Timestampable

Erinnern Sie sich noch an den Doctrine-Datentyp `datetime`, der in PHP auf ein `DateTime`-Objekt abgebildet wird? Nun wissen Sie auch endlich, was damit gemeint ist.

Häufig soll ein Datensatz in der Datenbank die Information enthalten, wann er angelegt oder zuletzt aktualisiert wurde. Laut physischem Datenmodell existiert in der Tabelle `articles` eine Spalte `created_at`, die das Erstellungsdatum des Datensatzes beinhalten soll. Die Umsetzung dieser Anforderung ist auch gar nicht so kompliziert, wie es sich im ersten Moment anhört. Dank des Doctrine-Datentyps `datetime` und der Gedmo Doctrine Extensions ist dies nämlich mit einfachen Annotationen schnell erledigt.

Beispiel

```php
<?php

namespace Entities;

use Doctrine\ORM\Mapping as ORM;
use Gedmo\Mapping\Annotation as Gedmo;
```

```php
use Webmasters\Doctrine\ORM\Util;

/**
 * @ORM\Entity
 * @ORM\Table(name="articles")
 */
class Article
{
    // gekuerztes Beispiel

    /**
     * @ORM\Column(name="created_at", type="datetime")
     * @Gedmo\Timestampable(on="create")
     */
    protected $createdAt;

    public function getCreatedAt()
    {
        return $this->createdAt;
    }

    // gekuerztes Beispiel
}
```

src/Entities/Article.php (Version)

Codebeispiel 68 *src/Entities/Article.php (Version)*

Für eine bessere Lesbarkeit wird in Zeile 6 der Alias `Gedmo` festgelegt. In Zeile 18 benutzen wir den Parameter `name`, da wir die lowerCamelCase-Vorgabe `$createdAt` für den Attributnamen (laut unseren Programmierrichtlinien) mit der Tabellenspalte `created_at` aus unserem physischen Datenmodell in Einklang bringen wollen. Theoretisch könnten wir auf diesen optionalen Parameter verzichten. Laut der Namenskonventionen von Doctrine entspräche dann die Benennung der Spalte der Schreibweise des Attributs (also `createdAt`). Der Annotation `@Gedmo\Timestampable` übergeben wir in Zeile 19 im Parameter `on`, wann der Datensatz mit einem neuen Wert versehen werden soll. Mit `create` geschieht dies beim Anlegen eines Datensatzes in der Datenbank und mit `update` bei der Aktualisierung des Datensatzes, aber natürlich nur, wenn dies mittels Doctrine geschieht.

Achtung

Bei Attributen mit Datumswerten in Form von `DateTime`-Objekten geben wir

bei der Attributs-Definition keine Default-Werte an (siehe Zeile 21)!

Einen Setter benötigt das Attribut `$createdAt` nicht, da der Wert dieser Spalte automatisch generiert werden soll. Der Getter liefert **nach dem Anlegen des Datensatzes** automatisch eine Instanz der Klasse `DateTime` zurück. Mit dieser Instanz können natürlich die in dieser Lektion besprochenen Methoden `DateTime#format()`, `DateTime#modify()` und `DateTime#diff()` genutzt werden.

Achtung

Merken Sie sich bereits jetzt, dass Doctrine einen bereits vorhandenen Datensatz nur aktualisiert, wenn sich mindestens ein Wert im dazugehörigen Objekt geändert hat. Ein mit `@Gedmo\Timestampable` gepflegter Wert wird erst nach dieser Prüfung angepasst. Deshalb muss noch mindestens ein weiterer Wert geändert werden, damit das Datum aktualisiert wird.

Probleme beim Umgang mit Datums-Attributen

Würden Sie eine leere Instanz der `Article`-Klasse erstellen und wollten Sie sofort den Inhalt des Attributs `$createdAt` ausgeben, so hätten Sie allerdings derzeit ein Problem. Da man zur Anzeige von Datumswerten immer auf ein Chaining mit `DateTime#format()` zurückgreifen muss, fehlt uns das passende Objekt im Attribut.

```
$article = new Article();
echo $article->getCreatedAt()->format('d.m.Y');
```

Chaining-Problem

Codebeispiel 69 *Chaining-Problem*

Das Beispiel scheitert mit der Meldung `Fatal error: Call to a member function format() on a non-object`. Bei einem Chaining sollten Sie also möglichst dafür sorgen, dass das benötigte Objekt zu jedem Zeitpunkt existiert.

In »Band 1: Grundlagen der OOP« haben Sie bereits einen Ansatz kennengelernt, um ein solches Problem zu lösen. Der dortige Ansatz mit einer Befüllung des Attributs im Konstruktor bereitete mir bei Tests mit `DateTime` allerdings Probleme, da es so etwas wie ein »leeres« `DateTime`-Objekt derzeit nicht gibt. Im Falle von Datumswerten hilft uns deswegen die Utility-Klasse `Webmasters\Doctrine\ORM\Util\DateTime`.

Beispiel

```
 1 <?php
 2
 3 namespace Entities;
 4
 5 use Doctrine\ORM\Mapping as ORM;
 6 use Gedmo\Mapping\Annotation as Gedmo;
 7 use Webmasters\Doctrine\ORM\Util;
 8
 9 /**
10  * @ORM\Entity
11  * @ORM\Table(name="articles")
12  */
13 class Article
14 {
15     // gekuerztes Beispiel
16
17     /**
18      * @ORM\Column(name="created_at", type="datetime")
19      * @Gedmo\Timestampable(on="create")
20      */
21     protected $createdAt;
22
23     public function getCreatedAt()
24     {
25         return new Util\DateTime($this->createdAt);
26     }
27
28     // gekuerztes Beispiel
29 }
```

src/Entities/Article.php (Version 2)

Codebeispiel 70 *src/Entities/Article.php (Version 2)*

In Zeile 25 übergeben wir den Inhalt des Attributs `$createdAt` als Parameter an den Konstruktur der Utility-Klasse. Es sind nun mehrere Varianten möglich:

1. Das Attribut enthält kein Datum (z. B. `null`, einen leeren String oder einen String, der nicht als Datum interpretiert werden kann). Im Gegensatz zur Original-Klasse wird hier kein aktuelles Datum verwendet und der ungültige Wert vom Konstruktor auch **nicht** mit einem Fehler quittiert, was später unseren Code im Zusammenhang mit Formulareingaben sehr einfach halten wird. Mehr dazu erfahren Sie in Lektion 12.

2. Das Attribut enthält einen String, der einem gültigen Datum (z. B. 1970-01-01 oder 01.01.1970) entspricht.

3. Das Attribut enthält eine Instanz der Original-Klasse `DateTime`.

4. Das Attribut enthält eine Instanz der Klasse `Webmasters\Doctrine\ORM\Util\DateTime`.

Im ersten Fall liefert ein Aufruf der Methode `format()` den Wert des ersten Konstruktor-Parameters zurück. Dies gilt natürlich auch, wenn dieser Wert ein leerer String ist. Ein Aufruf von `modify()` wird ignoriert und bei `diff()` wird ein `false` zurückgegeben. In den anderen drei Fällen können Sie alle hier vorgestellten DateTime-Methoden ganz normal verwenden.

10.4 Doctrine und die Nutzung von DateTime-Objekten

Mit `@Gedmo\Timestampable` können Sie nun zwar Attribute automatisiert mit dem aktuellen Zeitpunkt befüllen, doch was ist, wenn ein `datetime`-Attribut ganz normal mittels Getter und Setter benutzt werden soll?

Beispiel

```php
<?php

namespace Entities;

use Doctrine\ORM\Mapping as ORM;
use Gedmo\Mapping\Annotation as Gedmo;
use Webmasters\Doctrine\ORM\Util;
```

```
 8
 9 /**
10  * @ORM\Entity
11  * @ORM\Table(name="articles")
12  */
13 class Article
14 {
15     // gekuerztes Beispiel
16
17     /**
18      * @ORM\Column(name="publish_at", type="datetime")
19      */
20     protected $publishAt;
21
22     // gekuerztes Beispiel
23
24     public function getPublishAt()
25     {
26         return new Util\DateTime($this->publishAt);
27     }
28
29     public function setPublishAt($publishAt)
30     {
31         $this->publishAt = new Util\DateTime($publishAt);
32     }
33
34     // gekuerztes Beispiel
35 }
```

src/Entities/Article.php (Version 3)

Codebeispiel 71 *src/Entities/Article.php (Version 3)*

In den Gettern und Settern des Attributs `$publishAt` wurde dafür gesorgt, dass immer eine Instanz unserer `DateTime`-Klasse vorliegt und sich somit das Attribut einfacher auf Basis von Formular-Eingaben befüllen lässt.

```
$entry = array('publish_at' => 'xyz');
$article = new Article($entry);
echo $article->getPublishAt()->format('d.m.Y'); // gibt 'xyz' a
```

Gelöstes Chaining-Problem

Codebeispiel 72 *Gelöstes Chaining-Problem*

Es ist lediglich zu beachten, dass beim Befüllen des Attributs `publishAt` mittels `ArrayMapper#setData()` der Array-Schlüssel `publish_at` verwendet werden muss. Genauso wird übrigens bei der Nutzung von `ArrayMapper#toArray()` aus

dem Attribut `publishAt` wieder der Array-Schlüssel `publish_at`.

10.5 Zusammenfassung

Sie haben in dieser Lektion erfahren, worin die Besonderheiten des Doctrine-Datentyps `datetime` liegen, wie Sie diese Besonderheiten für sich nutzen und wie Sie manche Probleme bei der Nutzung umgehen können.

10.6 Testen Sie Ihr Wissen

1. Welche Alternative(n) zu der Unmenge von Datums- und Zeitfunktionen haben Sie in dieser Lektion kennengelernt?
2. Womit kann man automatisiert ein Erstellungs- oder Aktualisierungsdatum in einem Attribut pflegen?

10.7 Aufgaben zur Selbstkontrolle

Übung 24:

Erstellen Sie die restlichen Entity-Klassen unserer Beispiel-Datenbank aus Abschnitt 4.2, also `Article` und `User`. Implementieren Sie diese Tabellen isoliert und ignorieren Sie dabei für den Moment die Beziehungen zwischen den Tabellen (und somit auch die Fremdschlüssel bzw. die Zwischentabelle `tagging`). Nutzen Sie für das Attribut `$createdAt` die Annotation `@Gedmo\Timestampable`. Vergessen Sie nicht, mittels Annotationen die korrekten Tabellennamen bzw. Spaltennamen festzulegen und abschließend den Mini-Controller *setup.php* im Browser aufzurufen.

10.8 Optionale Aufgaben

Übung 25:

Der Mini-Controller *reset.php* soll nun auch noch einen Datensatz zur Tabelle `users` und einen zur Tabelle `articles` hinzufügen. Vergessen Sie nicht, das Truncate für diese beiden Tabellen zu ergänzen.

11 Datenbank-Beziehungen mit Doctrine

In dieser Lektion lernen Sie
• wie Sie 1:n-Beziehungen in Doctrine abbilden. • wie Sie n:m-Beziehungen in Doctrine abbilden. • wie Sie Ihre Anwendung durch JOINs performanter gestalten können. • welche Probleme sich beim Debugging ergeben können und wie man diese umgeht.

11.1Das Problem

Sie sind nun in der Lage, Doctrine für einzelne Tabellen zu nutzen. Allerdings kommen nur die einfachsten Anwendungen ohne Beziehungen in der Datenbank aus. Die Mehrzahl aller Anwendungen benötigen eine oder mehrere 1:n-Beziehungen oder sogar n:m-Beziehungen. Wobei jedoch letztere meiner persönlichen Erfahrung nach seltener anzutreffen sind, da man meistens ein zusätzliches Attribut wie beispielsweise `createdAt` oder `updatedAt` benötigt und deswegen unter Verwendung einer dritten Entity die n:m-Beziehung in zwei 1:n-Beziehungen zerlegt.

11.2 Beziehungen per Annotation definieren

Auch die Beziehungen zwischen Entity-Klassen definieren wir bei Doctrine mit Hilfe von Annotationen. Die Besonderheit bei den Beziehungen ist: **Sie müssen diese auf beiden Seiten der Beziehung definieren, also in beiden beteiligten Klassen.**

Doch bevor wir uns den eigentlichen Annotationen zuwenden, müssen wir zunächst ein paar Regeln zum Konzept der ***Owning Side*** (dt. Eigentümerseite) und ***Inverse Side*** (dt. Gegenseite) lernen.

Die Eigentümerseite ist die Seite, welche Doctrine betrachtet, um den Status einer Beziehung zu prüfen und somit zu ermitteln, ob eine Aktualisierung der Datenbankinhalte nötig ist.

- Die Eigentümerseite verweist in einer 1:n- und n:m-Beziehung auf ihre Gegenseite unter Verwendung des Parameters `inversedBy`. Dieser nennt als Wert den Namen des Attributs auf der Gegenseite der Beziehung.
- Die Gegenseite verweist in einer 1:n- und n:m-Beziehung auf ihre Eigentümerseite unter Verwendung des Parameters `mappedBy`. Dieser nennt als Wert den Namen des Attributs auf der Eigentümerseite der Beziehung.
- Der n-Teil (zu erkennen am Fremdschlüssel in der Tabelle) einer 1:n-Beziehung ist immer die Eigentümerseite, somit kann hier niemals der Parameter `mappedBy` angegeben werden.
- Bei einer n:m-Beziehung kann die Eigentümerseite der n- oder m-Teil sein. Man muss sich jedoch für eine Seite der Beziehung entscheiden.

> **Achtung**
>
> Wurden lediglich Änderungen an der Gegenseite einer Beziehung vorgenommen, so werden diese beim Flush ignoriert. Stellen Sie sicher, dass Sie immer beide Seiten aktualisieren (oder aktualisieren Sie zumindest die Eigentümerseite).

11.3 1:n-Beziehungen abbilden

Zwischen `User` und `Article` besteht eine klassische `1:n`-Beziehung, was man an dem Fremdschlüssel `user_id` in der Tabelle `articles` erkennt. Ein User kann also mehrere Articles haben, zu einem Article gehört jedoch immer nur maximal ein User.

Ein User hat mehrere Articles, was wir durch ein neues Attribut in der Klasse `User` namens `$articles` darstellen werden. Da es mehrere sein können,

verwenden wir den Plural. Umgekehrt hat ein Article maximal einen zugeordneten User, also fügen wir in der Klasse `Article` ein Attribut `$user` ein. Damit sind die Attribute an sich vorhanden. Jetzt teilen wir Doctrine durch Annotationen mit, dass diese beiden Attribute eine Beziehung darstellen.

> Beachten Sie, dass die Benennung der Beziehungs-Attribute von den Spalten in der Datenbank abweicht. Es gibt in beiden Klassen ein Attribut und keines davon ist identisch mit dem Fremdschlüssel. Allerdings erstellt uns Doctrine auf Basis der Annotationen den benötigten Fremdschlüssel in der Datenbank. Laut Namenskonventionen entspricht der Name dieser Spalte dem Attribut auf der Eigentümerseite (ohne `$`) mit dem Suffix `_id`.

11.3.1 Klasse User

In der Klasse `User` annotieren wir `$articles` mit `@ORM\OneToMany`, da wir uns im 1(One)-Teil der Beziehung befinden und uns auf viele (Many) `Article` beziehen.

Die Klasse `User` bildet die Gegenseite zur Eigentümerseite `Article`, deren Tabelle `articles` den Fremdschlüssel `user_id` erhalten soll. Die Annotation der Gegenseite erwartet zwei Parameter. Bei `targetEntity` geben wir den Klassennamen der Eigentümerseite an und bei `mappedBy` wird als Wert der Name des Attributs (ohne `$`) in der Klasse der Eigentümerseite erwartet.

Beispiel

```
/**
 * @ORM\OneToMany(targetEntity="Article", mappedBy="user")
 */
protected $articles;
```

src/Entities/User.php - Annotation auf der Gegenseite

Codebeispiel 73 *src/Entities/User.php - Annotation auf der Gegenseite*

Achten Sie bei den Werten von `targetEntity` und `mappedBy` bitte unbedingt auf die **korrekte Groß-/Kleinschreibung** der beiden Bezeichner. Beispielsweise darf der Wert von `mappedBy` (hier `user`) **niemals** mit einem Großbuchstaben

beginnen, da dies unseren Programmierrichtlinien für Attribute widerspricht.

> **Achtung**
>
> Bei Attributen für Beziehungen sollten Sie **niemals** einen Default-Wert bei der Definition angeben!

11.3.2 Klasse Article

Im n-Teil der Beziehung, also in der Klasse `Article`, benötigen wir die Annotation `@ORM\ManyToOne`. Diese Annotation der Eigentümerseite erwartet als Parameter `targetEntity` mit dem Klassennamen der Gegenseite und `inversedBy` mit dem dortigen Attributsnamen der Beziehung.

Beispiel

```
/**
 * @ORM\ManyToOne(targetEntity="User", inversedBy="articles")
 */
protected $user;
```

src/Entities/Article.php - Annotation auf der Eigentümerseite

Codebeispiel 74 *src/Entities/Article.php - Annotation auf der Eigentümerseite*

	Eigentümerseite	Gegenseite
Klasse	Article	User
Annotation	@ORM\ManyToOne	@ORM\OneToMany
Parameter	targetEntity="User" inversedBy="articles"	targetEntity="Article" mappedBy="user"

Tabelle 11.1 *Zusammenfassung*

> **Achtung**
>
> Bei `@ORM\OneToMany` (Gegenseite) heißt der zweite Parameter `mappedBy`, bei `@ORM\ManyToOne` (Eigentümerseite) jedoch `inversedBy`. Nicht verwechseln!

11.3.3 Das Problem

Doctrine hat nun zwar theoretisch Kenntnis von der 1:n-Beziehung, doch fehlen für den Zugriff auf jeden Fall noch ein paar Methoden.

> **Übung 26:**
>
> Überlegen Sie, welche wichtigen Methoden in `User` und `Article` derzeit noch fehlen.

Und? Haben Sie herausgefunden, welche Methoden in den beiden Klassen derzeit noch fehlen? Nein? Dann überlegen Sie noch einmal genau, welche Methoden für den Zugriff auf `protected`-Attribute nötig sind.

Ist der Groschen nun gefallen? Es fehlen natürlich die Getter und Setter für die beiden neuen Attribute.

> Im Fall des Attributs `$articles` liegt eine Besonderheit vor, die den Setter von allen bisherigen unterscheidet. Diese Besonderheit wird bereits durch den im Namen verwendeten Plural verdeutlicht. Unser Setter muss nämlich einen Article zu einer mehr oder weniger leeren **Sammlung** von Articles **ergänzen**.

Zunächst müssen Sie jedoch dafür sorgen, dass im Konstruktor immer für die Existenz einer Article-Sammlung (engl. collection) gesorgt wird.

Beispiel

```
public function __construct(array $data = array())
{
    Util\ArrayMapper::setEntity($this)->setData($data);
    $this->articles = new \Doctrine\Common\Collections\ArrayCol
}
```

src/Entities/User.php - Konstruktor mit einer Sammlung

Codebeispiel 75 *src/Entities/User.php - Konstruktor mit einer Sammlung*

Wie Sie sehen, wird hier eine Sammlung von Objekten durch ein Objekt der Klasse `Doctrine\Common\Collections\ArrayCollection` verwaltet. Um die Sammlung dann mit Daten zu befüllen oder vorhandene Daten zu bearbeiten, benötigen Sie neben dem normalen Getter eine Reihe von *Delegator*-Methoden (siehe »Band 1: Grundlagen der OOP«). Diese sind zwar nicht unbedingt verpflichtend, erleichtern jedoch den Umgang mit den Sammlungen.

clear

Beispiel

```
public function clearArticles()
{
    $this->articles->clear();
}
```

src/Entities/User.php - clear

Codebeispiel 76 *src/Entities/User.php - clear*

Oftmals ist es nötig, eine Sammlung komplett zu leeren. Hierzu dient die `clear`-Methode. Wir können diese Methode (und auch die nachfolgenden Methoden) allerdings nur deshalb verwenden, weil es sich beim Wert des Attributs `$articles` um ein `ArrayCollection`-Objekt und nicht um ein normales Array handelt.

add

Beispiel

```
public function addArticle(Article $article)
{
    $this->articles->add($article);
}
```

src/Entities/User.php - add

Codebeispiel 77 *src/Entities/User.php - add*

Sie befüllen mit diesem Setter kein komplettes Attribut, sondern ergänzen (engl. add) lediglich ein Objekt zu einer Sammlung. Diese Variante nennt man

deswegen auch ***Adder-Methoden*** oder kurz ***Adder***. Mit Type-Hinting wird sichergestellt, dass nur Objekte der Klasse `Article` hinzugefügt werden können.

contains

Beispiel

```
public function hasArticle(Article $article)
{
    return $this->articles->contains($article);
}
```

src/Entities/User.php - contains

Codebeispiel 78 *src/Entities/User.php - contains*

Vor allem beim Editieren von Datensätzen ist oftmals eine Prüfung nötig, ob ein Objekt überhaupt in einer Sammlung enthalten ist. Hierzu dient die `contains`-Methode.

removeElement

Beispiel

```
public function removeArticle(Article $article)
{
    $this->articles->removeElement($article);
}
```

src/Entities/User.php - removeElement

Codebeispiel 79 *src/Entities/User.php - removeElement*

Wenn Sie Objekte zu einer Sammlung ergänzen können, muss es natürlich auch eine Möglichkeit geben, ein bestimmtes Objekt wieder aus der Sammlung zu entfernen. Dies geschieht mit der `removeElement`-Methode.

> Das Attribut `$articles` wird zwar im Plural geschrieben, doch drei unserer Methoden hantieren lediglich mit einem einzigen Objekt `$article` als

Parameter. Aus diesem Grund habe ich bei diesen drei Methoden die Methoden-Namen im Singular formuliert. Unterscheiden sich Plural und Singular nicht, so ist dies jedoch nicht weiter problematisch.

11.3.4 Die Handhabung von 1:n-Beziehungen

Sie wissen nun, wie Sie schon vorhandene Klassen mittels Annotationen um 1:n-Beziehungen erweitern. Doch wie erstellt man eine Beziehung zwischen zwei Datensätzen bzw. wie löscht man eine solche Beziehung?

Hierzu müssen wir uns erst einmal klarmachen, dass wir eine sogenannte **bidirektionale Beziehung** verwenden wollen. Dies bedeutet, dass wir mittels Getter-Methoden jede der beiden Entities aus der jeweils anderen Entity bzw. dem dortigen Attribut für die Beziehung ansteuern können. Wir können also beispielsweise den User nach seinen Articles und umgekehrt auch jeden Article nach seinem User fragen. Dies bedeutet aber theoretisch auch, dass wir bei einer Anpassung der Beziehung die Attribute auf beiden Seiten aktualisieren müssen, damit die Entities nicht ihre Synchronizität verlieren.

Beziehung herstellen

Beispiel

```
$user->addArticle($article); // Gegenseite
$article->setUser($user); // Eigentuemerseite
$em->flush();
```

Bidirektionale Beziehung herstellen

Codebeispiel 80 *Bidirektionale Beziehung herstellen*

Beachten Sie, dass wir zunächst die Gegenseite aktualisiert haben und dann erst die Eigentümerseite. Ein `EntityManager#persist()` ist übrigens unnötig, sofern sich beide Entities bereits als Datensätze in der Datenbank befinden und lediglich eine Beziehung zwischen ihnen hergestellt wird.

Beziehung löschen

Beispiel

```
$user->removeArticle($article); // Gegenseite
$article->setUser(null); // Eigentuemerseite
$em->flush();
```

Bidirektionale Beziehung löschen

Codebeispiel 81 *Bidirektionale Beziehung löschen*

Zunächst wird wieder die Collection auf der Gegenseite aktualisiert und dann das Attribut auf der Eigentümerseite. Beachten Sie, dass es auf der Eigentümerseite keine spezielle Methode zum Löschen gibt, sondern dass `Article#setUser()` als Parameter `null` übergeben bekommt. Hierfür ist es wichtig, dass der Setter nicht mittels ***Type-Hinting*** ein Objekt der Klasse `User` verlangt. Sollte dies jedoch der Fall sein, so könnte man eine eigene Methode `unsetUser` (ohne Parameter) anlegen, die das Attribut auf `null` setzt. Außerdem bedeutet dies natürlich, dass die Fremdschlüssel-Spalte in der Datenbank im Gegensatz zu den restlichen Spalten standardmäßig leer sein darf. Beachten Sie außerdem, dass lediglich die Beziehung gelöscht wird. Beide Entities verbleiben als Datensätze in der Datenbank.

> Da Doctrine lediglich die Eigentümerseite auf Aktualisierungen untersucht, ist es für Schreiboperationen unnötig, die Sammlung auf der Gegenseite synchron zu halten. Indem Sie auf diese Synchronizität verzichten, verbessern Sie die Performance Ihrer Anwendung. Dies sollten Sie allerdings nur tun, wenn direkt danach eine Header-Umleitung stattfindet, da in diesem Fall in der Aktion zum Speichern keine Ausgabe und somit auch keine Getter-Zugriffe mehr erfolgen.
>
> Wir verzichten allerdings sowieso nur beim Speichern auf diese Synchronizität. Werden danach die Datensätze mittels Doctrine wieder aus der Datenbank ausgelesen, so sind die Getter-Zugriffe problemlos möglich. Wieso dies so ist, erfahren Sie am Ende dieser Lektion. Haben Sie bitte noch etwas Geduld.

Eine solche Header-Umleitung sollte jedoch sowieso bei jeder (oder zumindest jeder formularbasierten) Änderung von Datenbankinhalten erfolgen, um Probleme mit Browser-Reloads zu vermeiden. Mit letzterem Problem und vor

allem seiner Lösung werden wir uns übrigens noch einmal genauer in Lektion 12 beschäftigen.

Übung 27:

1. Ergänzen Sie in den Entity-Klassen `User` und `Article` den nötigen Code für die 1:n-Beziehung (Attribute, Annotationen, Getter und Setter bzw. Delegator-Methoden).
2. Rufen Sie den Controller *setup.php* im Browser auf, um damit die noch fehlende Fremdschlüsselspalte zu erstellen.

11.3.5 Ein Beispiel

Um mit der Beziehung ein wenig herumspielen zu können, benötigen wir zunächst angepasste Datensätze. Wir ergänzen hierzu einige Zeilen im Mini-Controller *reset.php*, den Sie schon aus den optionalen Aufgaben kennen.

Beispiel

```php
<?php

require_once 'inc/bootstrap.inc.php';

use Entities\Tag, Entities\User, Entities\Article;

// Schritt 1
$em->getConnection()->query('SET FOREIGN_KEY_CHECKS=0;');
$em->getConnection()->query('TRUNCATE TABLE tags;');
$em->getConnection()->query('TRUNCATE TABLE users;');
$em->getConnection()->query('TRUNCATE TABLE articles;');
$em->getConnection()->query('SET FOREIGN_KEY_CHECKS=1;');

// Schritt 2
$entries = array(
    array('title' => 'HTML'),
    array('title' => 'JavaScript'),
    array('title' => 'PHP')
);

foreach ($entries as $entry) {
    $tag = new Tag($entry);
```

```
23      $em->persist($tag);
24  }
25
26  $entry = array(
27      'email' => 'honk@example.com',
28      'password' => 'Du_kommst_hier_nicht_rein!'
29  );
30  $user = new User($entry);
31  $em->persist($user);
32
33  $entry = array(
34      'title' => 'Nur ein Test',
35      'teaser' => 'Dies ist nur ein Test ...',
36      'news' => 'Erster Absatz' . "\n\n" . 'Zweiter Absatz',
37      'publish_at' => 'now',
38  );
39  $article = new Article($entry);
40
41  $user->addArticle($article);
42  $article->setUser($user);
43
44  $em->persist($article);
45
46  $em->flush();
47
48  ?>
49  Die Datenbankinhalte wurden angepasst.
```

reset.php (Version 1)

Codebeispiel 82 *reset.php (Version 1)*

Diese Version der *reset.php* hat zwei Besonderheiten:

- In Zeile 8 müssen wir vor den Truncate-Befehlen die Fremdschlüsselprüfung deaktivieren und danach in Zeile 12 wieder aktivieren. Würden wir dies nicht tun, so würde das Truncate mit der Meldung `Fatal error: Uncaught exception 'PDOException' with message 'SQLSTATE[42000]: Syntax error or access violation: 1701 Cannot truncate a table referenced in a foreign key constraint` verweigert.
- In den Zeilen 41 und 42 stellen wir dann eine bidirektionale Beziehung zwischen dem `User`- und dem `Article`-Objekt her, die dann in den Zeilen 44/46 beim Speichern des `Article`-Objekts (der Eigentümerseite) ebenfalls persistiert wird.

Beispiel

```php
<?php

namespace Controllers;

use Entities\Article;

class IndexController extends AbstractBase
{
    public function indexAction()
    {
        $em = $this->getEntityManager();
        $articles = $em
            ->getRepository('Entities\Article')
            ->findAll()
        ;

        $this->addContext('articles', $articles);
    }

    // gekuerztes Beispiel
}
```

src/Controllers/IndexController.php

Codebeispiel 83 *src/Controllers/IndexController.php*

```php
<?php foreach ($articles as $article): ?>
    <article>
        <header>
            <h1><?php echo $article->getTitle(); ?></h1>
        </header>

        <div class="teaser">
            <?php echo $article->getTeaser(); ?>
        </div>

        <footer>
            Erstellt am
            <time datetime="<?php echo $article->getCreatedA
                <?php echo $article->getCreatedAt()->format(
            </time>
            von <?php echo $article->getUser(); ?>
        </footer>
    </article>
    <hr />
<?php endforeach; ?>
```

templates/IndexController/indexAction.tpl.php (Version 1)

Codebeispiel 84 *templates/IndexController/indexAction.tpl.php (Version 1)*

Sobald Sie mittels `EntityRepository#findAll()` alle Datensätze der Entity `Article` ermittelt haben, können Sie diese ganz normal im Template im Rahmen einer `foreach`-Schleife einzeln anzeigen. Von jedem Datensatz zeigen wir den Titel, den Teaser, den Erstellungszeitpunkt und die Mail-Adresse des Erstellers an. Für Letzteres nutzen wir die Methode `Article#getUser()`, welche uns das komplette `User`-Objekt zurückliefert.

Sie haben doch hoffentlich den Alternativschlüssel `email` in `User` als Rückgabewert der `__toString()`-Methode benutzt, oder haben Sie diese etwa komplett vergessen? Würden wir vom User beispielsweise den Wert des Attributs ID benötigen, so müssten wir stattdessen auf folgendes Chaining zurückgreifen `$article->getUser()->getId()`.

Übung 28:

1. Implementieren Sie den aktuellen Stand der *reset.php* in Ihrem Projekt. Rufen Sie anschließend den Mini-Controller im Browser auf.
2. Implementieren Sie den aktuellen Stand der *IndexController.php* und der *indexAction.tpl.php* in Ihrem Projekt.
3. Überprüfen Sie die komplette Implementierung durch einen Aufruf des Front-Controllers *index.php* im Browser.

11.4 n:m-Beziehungen abbilden

Kommen wir nun zur Königsdisziplin unter den hier vorgestellten Datenbank-Beziehungen, der n:m-Beziehung zwischen `Article` und `Tag` (zu erkennen an der Zwischentabelle `tagging`). Ein Article kann beliebig viele Tags haben und zu einem Tag gehören meist mehrere Articles.

Doch welcher Teil der Beziehung soll nun unsere Eigentümer- und welcher die Gegenseite sein? **Für n:m-Beziehungen liegt diese Entscheidung ja bei uns.** Aus Programmierersicht gibt es jedoch oftmals einen einfachen Anhaltspunkt für diese Entscheidung. Sie müssen sich nur fragen, welche Entity

die Verknüpfung handhabt, und diese als Eigentümerseite wählen.

Beispiel

Betrachten wir das sehr ähnlich gelagerte Blogsystem WordPress, das ebenfalls Artikel (Posts) und Schlagwörter (Tags) bietet. Immer wenn Sie einen Artikel mit einem Schlagwort verknüpfen wollen, managt der Artikel die Verknüpfung. Legt man nämlich einen neuen Artikel an, so erlaubt das entsprechende Formular die Auswahl bereits bestehender und das Anlegen neuer Schlagwörter, d.h. zuerst wird ein Schlagwort in der Datenbank gespeichert und erst danach kann es mit einem Artikel verknüpft werden. Außerdem aktualisiert man häufiger den einen oder anderen Artikel, aber so gut wie nie ein bestehendes Schlagwort. Aus diesem Grund würde man den Artikel als Eigentümerseite wählen.

11.4.1 Klasse Article und Klasse Tag

In unserem Newsticker könnte sich ein bereits existierender Benutzer einloggen, um danach Tags oder Articles anzulegen. Wenn er neue Tags anlegt, so würde er dies vermutlich ohne direkte Zuordnung zu Articles tun. Dies wäre also äquivalent zu unserer derzeitigen Vorgehensweise in der *reset.php*. Erst danach würde der Benutzer einen neuen Article anlegen und dabei auswählen, welche der bereits existierenden Tags dem neuen Article zugeordnet werden sollen. Danach würden wir den Article speichern.

Im aktuellen Fall ist es also sinnvoll, die Entity `Article` als **Eigentümerseite** und die Entity `Tag` als **Gegenseite** festzulegen. In der Klasse `Article` ergänzen wir zunächst ein Attribut `$tags` und in `Tag` nennen wir das Gegenstück `$articles`, wobei wir bei beiden Bezeichnern den Plural verwenden. Danach versehen wir beide Attribute mit einer `@ORM\ManyToMany`-Annotation.

Beispiel

```
/**
 * @ORM\ManyToMany(targetEntity="Tag", inversedBy="articles")
 */
```

```
protected $tags;
```

src/Entities/Article.php - Annotation auf der Eigentümerseite

Codebeispiel 85 *src/Entities/Article.php - Annotation auf der Eigentümerseite*

```
/**
 * @ORM\ManyToMany(targetEntity="Article", mappedBy="tags")
 */
protected $articles;
```

src/Entities/Tag.php - Annotation auf der Gegenseite

Codebeispiel 86 *src/Entities/Tag.php - Annotation auf der Gegenseite*

	Eigentümerseite	Gegenseite
Klasse	Article	Tag
Annotation	@ORM\ManyToMany	@ORM\ManyToMany
Parameter	targetEntity="Tag" inversedBy="articles"	targetEntity="Article" mappedBy="tags"

Tabelle 11.2 *Zusammenfassung*

> Bei einer n:m-Beziehung muss in beiden Klassen im Konstruktor eine Sammlung für das entsprechende Attribut initialisiert werden. Für jede dieser Sammlungen benötigen wir anstatt eines Setters die vier Delegator-Methoden.

11.4.2 Die Zwischentabelle

Für die n:m-Beziehung würde Doctrine auf Basis der Namenskonventionen die Zwischentabelle `article_tag` anlegen. Die Konvention besagt, dass der Tabellenname aus den beiden Klassennamen (ohne Namespace) und einem Unterstrich als Trenner gebildet wird. Genauso werden die Spaltennamen in der Zwischentabelle standardmäßig aus den beiden Klassennamen gefolgt von einem `_id` gebildet. Beachten Sie bitte, dass bei beiden Konventionen eine Umwandlung in Kleinbuchstaben erfolgt.

Wenn Ihnen die Namen egal sind, reduziert sich der Mapping-Code also auf ein

Minimum. In unserem Fall entspricht der Name der Zwischentabelle jedoch nicht der Datenbankplanung, wo als Name `tagging` vorgegeben wurde. Diese Änderung erreichen wir ganz einfach mit der Annotation `@ORM\JoinTable`, der wir als Parameter den gewünschten Tabellennamen übergeben können und die auf der Eigentümerseite ergänzt wird.

Beispiel

```
/**
 * @ORM\ManyToMany(targetEntity="Tag", inversedBy="articles")
 * @ORM\JoinTable(name="tagging")
 */
protected $tags;
```

src/Entities/Article.php - Annotation auf der Eigentümerseite

Codebeispiel 87 *src/Entities/Article.php - Annotation auf der Eigentümerseite*

```
/**
 * @ORM\ManyToMany(targetEntity="Article", mappedBy="tags")
 */
protected $articles;
```

src/Entities/Tag.php - Annotation auf der Gegenseite

Codebeispiel 88 *src/Entities/Tag.php - Annotation auf der Gegenseite*

Übung 29:

1. Ergänzen Sie in den Entity-Klassen `Article` und `Tag` die fehlenden Attribute und Annotationen für die n:m-Beziehung.
2. Beide Entity-Klassen benötigen für die neuen Attribute die vier bekannten Delegator-Methoden. Setzen Sie diese inklusive der beiden Getter um.
3. Rufen Sie abschließend den Mini-Controller *setup.php* im Browser auf, um damit die noch fehlende Zwischentabelle zu erstellen.

11.4.3 Ein Beispiel

Um mit der n:m-Beziehung ein wenig herumspielen zu können, müssen wir

erneut den Mini-Controller *reset.php* anpassen.

Beispiel

```php
<?php

require_once 'inc/bootstrap.inc.php';

use Entities\Tag, Entities\User, Entities\Article;

// Schritt 1
$em->getConnection()->query('SET FOREIGN_KEY_CHECKS=0;');
$em->getConnection()->query('TRUNCATE TABLE tags;');
$em->getConnection()->query('TRUNCATE TABLE users;');
$em->getConnection()->query('TRUNCATE TABLE articles;');
$em->getConnection()->query('TRUNCATE TABLE tagging;');
$em->getConnection()->query('SET FOREIGN_KEY_CHECKS=1;');

// Schritt 2
$entries = array(
    array('title' => 'HTML'),
    array('title' => 'JavaScript'),
    array('title' => 'PHP')
);

foreach ($entries as $entry) {
    $tag = new Tag($entry);
    $em->persist($tag);
}

$entry = array(
    'email' => 'honk@example.com',
    'password' => 'Du_kommst_hier_nicht_rein!'
);
$user = new User($entry);
$em->persist($user);

$em->flush();

// Schritt 3
$query = $em
    ->createQueryBuilder()
    ->select('t')
    ->from('Entities\Tag', 't')
    ->where('t.title = :title1 OR t.title = :title2')
    ->setParameters(array('title1' => 'HTML', 'title2' => 'P
    ->getQuery()
;
```

```
45 $tags = $query->getResult();
46
47 $entry = array(
48     'title' => 'Nur ein Test',
49     'teaser' => 'Dies ist nur ein Test ...',
50     'news' => 'Erster Absatz' . "\n\n" . 'Zweiter Absatz',
51     'publish_at' => 'now',
52 );
53 $article = new Article($entry);
54
55 $user->addArticle($article);
56 $article->setUser($user);
57
58 foreach ($tags as $tag) {
59     $tag->addArticle($article);
60     $article->addTag($tag);
61 }
62
63 $em->persist($article);
64
65 $em->flush();
66
67 ?>
68 Die Datenbankinhalte wurden angepasst.
```

reset.php (Version 2)

Codebeispiel 89 *reset.php (Version 2)*

Was hat sich im Controller geändert?

- In Zeile 12 wurde ein Truncate für die Tabelle `tagging` ergänzt.
- In Zeile 34 wurde ein Aufruf von `EntityManager#flush()` ergänzt.
- In den Zeilen 37 bis 45 lesen wir zwei der drei Tags wieder aus der Datenbank aus. Dies wäre ohne das vorherige Flush (noch) nicht möglich, da die eigentliche Speicherung ja erst durch dieses Flush stattfindet.
- In Zeile 41 habe ich zwei `WHERE`-Bedingungen direkt mit `OR` verknüpft. Die Methode `QueryBuilder#orWhere()` ist hierfür also nicht zwingend nötig, verbessert aber oftmals die Lesbarkeit des Codes.
- Die Zeilen 58 bis 61 dienen dann dazu, die Beziehung zwischen diesen beiden Tags und unserem Article herzustellen.

Auch das Template *indexAction.tpl.php* muss erneut angepasst werden.

Beispiel

```
<?php foreach ($articles as $article): ?>
    <article>
        <header>
            <h1><?php echo $article->getTitle(); ?></h1>
        </header>

        <div class="teaser">
            <?php echo $article->getTeaser(); ?>
        </div>

        <footer>
            Erstellt am
            <time datetime="<?php echo $article->getCreatedA
                <?php echo $article->getCreatedAt()->format(
            </time>
            von <?php echo $article->getUser(); ?>
            <br />
            Tags:
            <?php echo implode(' ', $article->getTags()->toA
        </footer>
    </article>
    <hr />
<?php endforeach; ?>
```

templates/IndexController/indexAction.tpl.php (Version 2)

Codebeispiel 90 *templates/IndexController/indexAction.tpl.php (Version 2)*

Wie Sie sehen, wurde in Zeile 17 ein Zeilenumbruch ergänzt. In Zeile 19 lesen wir dann das Attribut `$tags` unseres `Article`-Objekts aus. Hierbei handelt es sich um eine Instanz der Klasse `ArrayCollection`. Diese kann leider nicht direkt als Parameter an die Funktion `implode()` übergeben werden, da hier ein Array erwartet wird. Deswegen wird an den Getter die Methode `ArrayCollection#toArray()` angekettet.

Sie kennen nun also fünf Methoden, die Ihnen durch die Klasse `ArrayCollection` zur Verfügung stehen, nämlich `ArrayCollection#clear()`, `ArrayCollection#add()`, `ArrayCollection#contains()`, `ArrayCollection#removeElement()` und `ArrayCollection#toArray()`. Merken Sie sich bitte außerdem noch die beiden äußerst nützlichen Methoden `ArrayCollection#isEmpty()` und `ArrayCollection#count()`, die sich mit der Anzahl der Objekte in der Collection befassen.

Übung 30:

1. Implementieren Sie den aktuellen Stand der *reset.php* in Ihrem Projekt. Rufen Sie den Mini-Controller anschließend im Browser auf.
2. Implementieren Sie den aktuellen Stand der *indexAction.tpl.php* in Ihrem Projekt.
3. Überprüfen Sie die komplette Implementierung durch einen Aufruf des Front-Controllers *index.php* im Browser. Werden die beiden Tags des Article angezeigt?

11.5 Lazy Loading

Rekapitulieren wir noch einmal kurz die grundsätzliche Vorgehensweise. Das Hinzufügen eines Tags zu einem Article geschieht, indem wir `Article#addTag()` ein `Tag`-Objekt übergeben und danach die Methode `EntityManager#flush()` aufrufen. Wenn Sie Wert auf eine Synchronizität der bidirektionalen Beziehung legen, so müssen Sie natürlich zuallererst noch `Tag#addArticle()` das `Article`-Objekt übergeben. Den Rest, sprich das Hinzufügen des Eintrags in der Zwischentabelle, erledigt Doctrine für uns. Auch hier müssen Sie sich somit nicht mit SQL herumschlagen.

Auch das Auslesen im Template, wenn wir den User oder die Tags eines Article anzeigen wollen, geschieht automatisch, sobald wir auf das entsprechende Attribut zugreifen. In unserem Code wurde beispielsweise nach dem User gefragt, wenn wir im Template die Methode `Article#getUser()` aufgerufen haben. Damit dieser Abruf von zusätzlichen Daten funktionieren kann, benutzt Doctrine ein sogenanntes ***Lazy Loading*** (auch Deferred Loading oder Delayed Loading genannt, dt. verspätetes Laden). Lazy Loading ist übrigens das Gegenteil zum Eager Loading (alias Immediate Loading, dt. sofortiges Laden).

Betrachten wir das nachfolgende Beispiel:

Beispiel

```php
<?php

require_once 'inc/bootstrap.inc.php';

$article = $em
    ->getRepository('Entities\Article')
    ->find(1)
;
$user = $article->getUser();

?>
Klasse: <?php echo get_class($user); ?>
<br />
Elternklasse: <?php echo get_parent_class($user); ?>
<br />

<?php if ($user instanceof Entities\User): ?>
    $user ist eine Instanz der Elternklasse Entities\User.
<?php endif; ?>
```

test_lazy_loading.php

Codebeispiel 91 *test_lazy_loading.php*

Dies ist ein weiterer Mini-Controller, in welchem wir ein paar kleine Tests zum ***Lazy Loading*** durchführen. Zunächst lesen wir den Article mit der ID 1 aus und in Zeile 9 benutzen wir dann `Article#getUser()`, um den zugehörigen User zu ermitteln. Auf diesem Weg ist es theoretisch möglich, eine schier unendliche Kette von Beziehungsabfragen über Getter aufzubauen. Praktisch wird dies jedoch (wie so oft) vom Arbeitsspeicher des Webservers verhindert.

Unser Data Mapper Doctrine muss nun dafür sorgen, dass ein schon geladenes Objekt problemlos auf das nachfolgende in einem solchen Beziehungsgeflecht zugreifen kann, und dies, ohne uns als Programmierer mit dem Aufruf von zusätzlichen Finder-Methoden zu belasten. Hierzu kommen anstatt direkter Instanzen unserer Entity-Klassen sogenannte ***Proxy-Objekte*** zum Einsatz. Diese Proxy-Objekte sind Instanzen von durch Doctrine automatisch generierten ***Proxy-Klassen***.

```
Klasse: Proxies\__CG__\Entities\User
Elternklasse: Entities\User
$user ist eine Instanz der Elternklasse Entities\User.
```

test_lazy_loading.php (Ausgabe)

Codebeispiel 92 *test_lazy_loading.php (Ausgabe)*

Wie wir in der Ausgabe unsereres Test-Controllers erkennen, hat die Proxy-Klasse `User` in diesem Fall den Namespace `Proxies\__CG__\Entities`. Mittels Vererbung leitet sich diese Klasse von der ursprünglichen Entity-Klasse ab. Durch diese Vorgehensweise ist ein `instanceof` weiterhin erfolgreich. Innerhalb der Proxy-Klasse hat Doctrine jedoch jede einzelne unserer öffentlichen Entity-Methoden mit eigenen Anweisungen erweitert. Im Moment unseres Zugriffs auf eine dieser Methoden kann deshalb das Proxy-Objekt (mittels einer Datenbank-Abfrage) die benötigten Daten ermitteln und so die Anzeige einer Fehlermeldung vermeiden.

Die Bootstrap-Klasse der Webmasters Doctrine Extensions aktiviert übrigens die automatische Erstellung dieser Klassen im Temp-Ordner unseres Webservers, welchen Sie mit der Anweisung `ini_get('session.save_path')` in Erfahrung bringen können.

11.6 JOINs

Kommen wir nun zurück zu unseren Datenbank-Beziehungen und dem dort benutzten Lazy Loading. Bei vier Datensätzen und zwei Beziehungen ist das noch kein Problem, aber stellen Sie sich vor, Sie listen 50 Datensätze auf und jeder hat vier unterschiedliche Beziehungen. Das sind im schlimmsten Fall mehrere hundert Datenbank-Anfragen, die Sie zur Ermittlung der zusätzlichen Daten benötigen! Zur Vermeidung dieses Problems kann es Sinn machen, die abhängigen Datensätze gleich in einem Rutsch mit den Articles zu laden. Dies erreichen wir über JOINs, die es auch in Doctrine gibt. So haben wir im besten Fall nur ein einziges, großes SQL- bzw. DQL-Statement, das alle Daten auf einmal lädt, egal um wie viele Datensätze es sich dabei handelt.

Bevor wir die Implementierung in Doctrine betrachten, möchte ich Sie bitten, vorher noch Ihr Wissen über Datenbank-JOINs aufzufrischen. Sie werden diese Kenntnisse nämlich brauchen. Lesen Sie bitte erst weiter, wenn Sie (wieder) wissen, was ein ***JOIN*** ist und wie er funktioniert. Das ist eine bewusste Unterstellung, denn auch ich selbst muss nach einer längeren Pause jedes Mal wieder nachschlagen, wie die Syntax genau lautet.

Mit den normalen Finder-Methoden können wir übrigens keine JOINs durchführen, dafür benötigen wir DQL oder den `QueryBuilder`.

11.6.1 Per DQL

Die Syntax bei einem Doctrine-JOIN unterscheidet sich leicht vom klassischen SQL-JOIN. Sie müssen nicht angeben, auf welche Spalten der JOIN durchgeführt werden soll, da Doctrine die Beziehungen bereits durch die Annotationen kennt. Stattdessen sagen Sie einfach, welches Attribut per JOIN eingebunden werden soll. Auch die Art der Beziehung spielt keine Rolle, eine 1:n-Beziehung wird exakt so eingebunden wie eine n:m-Beziehung.

Für die meisten Fälle werden Sie nur den `LEFT JOIN` benötigen. Ein `LEFT JOIN`, der in den Articles gleich die Users einbindet, sieht in DQL so aus:

Beispiel

```
SELECT a, u FROM Entities\Article a LEFT JOIN a.user u
```

Wir sagen also nur: Mach einen JOIN auf das Attribut `$user` und benutze für diesen JOIN bzw. die gejointe Entity den Alias `u`. Außerdem ergänzen wir diesen zweiten Alias hinter dem `SELECT`, d.h. wir listen hier alle Aliase der gejointen Entities durch Komma separiert auf.

Fertig im Controller sieht das Ganze so aus:

Beispiel

```
$query = $em->createQuery(
    "SELECT a, u FROM Entities\Article a LEFT JOIN a.user u"
);
$articles = $query->getResult();
```

Per DQL

Codebeispiel 93 *Per DQL*

11.6.2 Per QueryBuilder

Einen LEFT JOIN führt man im QueryBuilder mit der Methode `leftJoin()` aus. Der Aufbau ist ansonsten identisch zum DQL-Beispiel:

Beispiel

```
$query = $em
    ->createQueryBuilder()
    ->select('a, u')
    ->from('Entities\Article', 'a')
    ->leftJoin('a.user', 'u')
    ->getQuery()
;
$articles = $query->getResult();
```

Per QueryBuilder

Codebeispiel 94 *Per QueryBuilder*

11.7 Debugging-Probleme

Öfters will man sich vor dem Erstellen eines Templates die Werte in einem Objekt kurz ansehen. Vielleicht hat der eine oder andere dies im Rahmen dieser Lektion auch schon mit `var_dump()` probiert.

Beispiel

```
$query = $em
    ->createQueryBuilder()
    ->select('a, u')
    ->from('Entities\Article', 'a')
    ->leftJoin('a.user', 'u')
    ->getQuery()
;
$articles = $query->getResult();

var_dump($articles);
```

Problem mit var_dump()

Codebeispiel 95 *Problem mit var_dump()*

Führen Sie diesen Code besser nicht aus! Er wird nicht zum gewünschten Resultat führen. Stattdessen wird entweder ein extrem langer Dump ausgegeben, oder der Browser friert ein und stürzt ab. Alternativ könnten dem Webserver auch die Ressourcen ausgehen, was normalerweise in einem unbehebbaren Fehler endet (z. B. `Fatal error: Allowed memory size (...) exhausted` oder `Fatal error: Maximum execution time (...) exceeded`). Doch woran liegt dies?

Unser Dump enthält mindestens ein `Article`-Objekt. Aufgrund des JOINs enthält dieses Objekt im Attribut `$user` ein `User`-Objekt. Wir nutzen jedoch bidirektionale Beziehungen. Somit enthält das `User`-Objekt auch unser `Article`-Objekt bzw. genauer gesagt eine `ArrayCollection` mit `Article`-Objekten, welche wiederum `User`-Objekte enthalten...

Dabei habe ich das Tagging noch gar nicht berücksichtigt. Denn ein `Article`-Objekt enthält eine `ArrayCollection` mit `Tag`-Objekten, womit ein zweiter Kreislauf beginnt.

Ein vergleichbares Verhalten kennen Sie von Endlos-Schleifen, d.h. ein Skript befindet sich in einem endlosen Kreislauf, aus dem der PHP-Interpreter nicht mehr ausbrechen kann, da die Abbruchbedingung niemals `true` wird. Es gibt jedoch eine Alternative zu `var_dump()`, mit der wir eine solche Abbruchbedingung festlegen können.

Beispiel

```
$query = $em
    ->createQueryBuilder()
    ->select('a, u')
    ->from('Entities\Article', 'a')
    ->leftJoin('a.user', 'u')
    ->getQuery()
;
$articles = $query->getResult();

\Doctrine\Common\Util\Debug::dump($articles);
```

Debug::dump()

Codebeispiel 96 *Debug::dump()*

Die Klasse `Debug` im Namespace `Doctrine\Common\Util` hat eine statische Methode `dump`, mit der wir den Inhalt der Variablen `$articles` betrachten können. Wenn wir uns diese Ausgabe genauer ansehen, so erfahren wir, dass `$articles` ein numerische Array mit derzeit genau einem Wert ist. Dieser Wert ist ein Objekt der Klasse `Entities\Article`.

Alle Attribute dieses Objekts, die einen ***skalaren Datentyp*** (z. B. `string` oder `integer`) haben, werden normal ausgegeben. Als Wert für das Attribut `$user` wird jedoch einfach nur `Entities\User` und als Wert für `$tags` lediglich `Array(2)` angegeben. Dies liegt an besagter Abbruchbedingung. Standardmäßig darf die Methode nämlich nur maximal zwei Ebenen der gewünschten Variable ausgeben. Die erste Ebene bildet das numerische Array. Die zweite Ebene beginnt mit dem `Article`-Objekt. Jede weitere Ebene beginnt mit einem Array, einer ArrayCollection oder einem sonstigen Objekt. Die dritte Ebene wäre dann somit in dem `Article`-Objekt das `User`-Objekt im Attribut `$user` bzw. die `ArrayCollection` im Attribut `$tags`.

> **Achtung**
>
> Im Dump wird nicht zwischen Arrays und ArrayCollections unterschieden. Beides wird als `array` angezeigt.

Möchten wir die besagte dritte Ebene sehen, so müssen wir die Abbruchbedingung als optionalen zweiten Parameter `$maxDepth` explizit angeben.

Beispiel

```
\Doctrine\Common\Util\Debug::dump($articles, 3);
```

$maxDepth von 3

Codebeispiel 97 *$maxDepth von 3*

Nun sehen wir zusätzlich, welche skalaren Werte im `User`-Objekt vorhanden sind und welche Klasse die Objekte im Attribut `$tags` haben (nämlich `Entities\Tag`). Würden wir auch noch die skalaren Werte in den `Tag`-Objekten

sehen wollen, so müssten wir die maximale Tiefe auf `4` erhöhen.

Standardmäßig werden übrigens automatisch HTML-Tags mittels `strip_tags()` entfernt. Sollte dies nicht gewünscht sein, so müssen Sie den optionalen dritten Parameter `$stripTags` auf `false` setzen.

Beispiel

```
\Doctrine\Common\Util\Debug::dump($articles, 3, false);
```

strip_tags() deaktivieren

Codebeispiel 98 *strip_tags() deaktivieren*

HTML-Tags werden nun allerdings als HTML-Code interpretiert. Enthält der Wert eines Attributs also beispielsweise einen String mit einem `strong`-Tag, so wird der darin eingeschlossene Text in Fettschrift dargestellt.

11.8 Zusammenfassung

Sie sind nun in der Lage, 1:n- und n:m-Beziehungen mittels Annotationen umzusetzen und wissen, welche sonstigen Änderungen Ihre Entities benötigen. Ihnen ist klar, dass wir bidirektionale Beziehungen benutzen und dass bei deren Aktualisierung eine Anpassung der Eigentümerseite verpflichtend ist.

Außerdem wissen Sie, was Lazy Loading ist und wozu es benutzt wird. JOINs machen Ihre Datenbank-Abfragen und damit auch den PHP-Code nun zwar komplexer, Sie sparen jedoch einige ausgeführte SQL-Queries (besonders bei längeren Listen von Datensätzen). Sie müssen von Fall zu Fall selbst entscheiden, wann Ihnen einfacherer Code und wann Performance wichtiger ist.

Die Wahl zwischen Lazy Loading und Eager Loading entspricht übrigens oft der Entscheidung zwischen Pest und Cholera: Wenn man nicht genau weiß, ob die zusätzlichen Daten häufig benötigt werden, dann kann es ungünstig sein, sie sofort zu laden. Es kann aber auch problematisch sein, sie später nachladen zu müssen, wenn man hierfür viel Code im Controller anpassen muss. Mögliche Entscheidungskriterien sind:

- Wie wahrscheinlich ist es, dass die Daten benötigt werden?
- Wie groß ist die zusätzlich geladene Datenmenge?
- Wie schnell ist der aktuelle Seitenaufbau des Controllers?

Als Faustregel bezüglich JOINs kann ich Folgendes sagen: Wenn der Controller ohnehin schnell lädt und die Daten selten benötigt werden, wozu sich die Mühe machen? Sobald Sie bemerken, dass der Seitenaufbau schleppend vorangeht oder Sie die zusätzlichen Daten ständig benötigen, nutzen Sie JOINs.

11.9 Testen Sie Ihr Wissen

1. Wie definiert man Beziehungen zwischen Doctrine-Entities?
2. Genauer gefragt, was nutzt man, um eine 1:n-Beziehung abzubilden?
3. Und was nutzt man, um eine n:m-Beziehung abzubilden?
4. Welche Methoden der Klasse `ArrayCollection` haben Sie kennengelernt?
5. Was ist Lazy Loading?

11.10 Aufgaben zur Selbstkontrolle

Übung 31:

Überarbeiten Sie den Inhalt der Standard-Aktion `indexAction` im Standard-Controller `IndexController`. Ermitteln Sie nun **mittels QueryBuilder** alle Articles und nutzen Sie einen `LEFT JOIN`, um die Tags direkt mit auszulesen.

11.11 Optionale Aufgaben

Übung 32:

Fügen Sie zusätzlich auch noch den `LEFT JOIN` für die User zu Ihrer Abfrage hinzu.

12 Controller-Klassen im Überblick

In dieser Lektion lernen Sie
• wie Sie gängige Datenbankoperationen in Controller-Klassen abbilden können. • wie Sie bestehende Datensätze aktualisieren. • wie Sie bestehende Datensätze löschen.

12.1Einleitung

Sie wissen inzwischen, wie Sie mit Doctrine Datensätze aus der Datenbank auslesen und wie Sie neue Datensätze speichern können, doch noch fehlen Ihnen die Möglichkeiten, bestehende Datensätze zu aktualisieren oder gar zu löschen. Auch wäre es praktisch, wenn Sie zum Anlegen neuer Datensätze nicht immer die Werte in Ihrem Code hinterlegen müssten. Um all dies zu erreichen, sollten wir nun unsere Controller-Klassen überarbeiten.

12.2 Flash-Notices

Doch bevor wir zu den Änderungen kommen, müssen wir uns zunächst mit ein paar Methoden in der Klasse `AbstractBase` vertraut machen.

Beispiel

```php
<?php

namespace Controllers;

use Doctrine\ORM\EntityManager;

abstract class AbstractBase
{
    // gekuerztes Beispiel

```

```php
    protected function setMessage($message)
    {
        $_SESSION['message'] = $message; // Set flash message
    }

    protected function getMessage()
    {
        $message = false;
        if (isset($_SESSION['message'])) {
            // Read and delete flash message from session
            $message = $_SESSION['message'];
            unset($_SESSION['message']);
        }

        return $message;
    }

    // gekuerztes Beispiel

    protected function redirect($action = null, $controller =
    {
        $params = array();

        if (!empty($controller)) {
            $params[] = 'controller=' . $controller;
        }

        if (!empty($action)) {
            $params[] = 'action=' . $action;
        }

        $to = '';
        if (!empty($params)) {
            $to = '?' . implode('&', $params);
        }

        header('Location: index.php' . $to);
        exit;
    }

    protected function render()
    {
        extract($this->context);

        $message = $this->getMessage(); // Get flash message
        $template = $this->getTemplate();

        require_once $this->basePath . '/templates/layout.tpl
    }
}
```

src/Entities/AbstractBase.php

Codebeispiel 99 *src/Entities/AbstractBase.php*

Die Methode `AbstractBase#redirect()` dient dazu, Ihnen Schreibarbeit bei den doch relativ häufig verwendeten Header-Umleitungen zu ersparen. Als optionale Parameter werden lediglich die Werte für die URL-Parameter `controller` und `action` erwartet, der Front-Controller *index.php* und die beiden Bezeichner für die URL-Parameter werden automatisch ergänzt.

Die Methode `AbstractBase#setMessage()` ermöglicht es Ihnen, vor einer solchen Header-Umleitung eine Meldung in der Session abzulegen. Im Folgenden wird dies für die Erfolgsmeldungen der Aktionen genutzt.

Die Methode `AbstractBase#getMessage()` wird dann anschließend genutzt, um diese Meldung wieder aus der Session auszulesen. Dies geschieht automatisch in der Methode `AbstractBase#render()` in Zeile 55. Da jede Meldung nur einmalig angezeigt werden darf, erfolgt in Zeile 22 ein sofortiges Löschen der Meldung aus der Session mit `unset()`.

Beispiel

```
 1 <!DOCTYPE html>
 2 <html>
 3
 4 <head>
 5     <meta charset="utf-8" />
 6     <title>Newsticker</title>
 7     <link type="text/css" rel="stylesheet" href="css/styleshe
 8 </head>
 9
10 <body>
11     <header>
12         <h1>Newsticker</h1>
13     </header>
14
15     <?php require 'navi.tpl.php'; ?>
16     <?php require 'flash_message.tpl.php'; ?>
17     <?php require 'errors.tpl.php'; ?>
18
19     <section>
20         <?php require $template; ?>
21     </section>
22 </body>
```

```
23 </html>
```

templates/layout.tpl.php

Codebeispiel 100 *templates/layout.tpl.php*

```
1 <?php if ($message): ?>
2     <p class="message"><?php echo $message; ?></p>
3 <?php endif; ?>
```

templates/flash_message.tpl.php

Codebeispiel 101 *templates/flash_message.tpl.php*

Die Ausgabe von `$message` erfolgt dann im Partial *flash_message.tpl.php*, welches in Zeile 16 des Templates *layout.tpl.php* eingebunden wird. Das Layout-Template regelt außerdem die Ausgabe unserer zukünftigen Validierungsfehlermeldungen, indem es in Zeile 17 das Partial *errors.tpl.php* einbindet.

Das Prinzip (Nachricht in der Session zwischenspeichern, Header-Umleitung und dann Nachricht zur einmaligen Anzeige wieder auslesen) nennt man übrigens ***Flash-Notices*** (dt. aufblitzende Nachrichten).

12.3 BREAD

Das Akronym ***CRUD*** ist Ihnen eventuell schon bekannt. Von manchen Autoren wird auch alternativ das Akronym ***RUDI*** (**I**nsert an Stelle von Create) oder ***CDUR*** (in Anlehnung an die Tonart) benutzt. ***CRUD*** umfasst die grundlegenden Datenbankoperationen **C**reate (Datensatz anlegen), **R**ead/Retrieve (Datensatz bzw. Datensätze auslesen und anzeigen), **U**pdate (Datensatz aktualisieren) und **D**elete/Destroy (Datensatz löschen).

Bei der Umsetzung von Benutzeroberflächen bei MVC-basierten Anwendungen haben wir mit CRUD jedoch ein Problem, da sich der Controller- und der Template-Code zur Anzeige eines bzw. mehrerer Datensätze in wesentlichen Punkten unterscheidet. Dieser wichtige Unterschied wird von ***BREAD*** (ein Akronym für **B**rowse (Datensätze auflisten), **R**ead (Datensatz auslesen und anzeigen), **E**dit (Datensatz aktualisieren), **A**dd (Datensatz anlegen) und **D**elete) besser berücksichtigt.

Doch wie könnte eine solche Umsetzung aussehen? Um dies herauszufinden, werden wir in unserer Controller-Klasse `IndexController` für jede BREAD-Operation von `Article` eine Methode erarbeiten. Außerdem werden wir uns die eine oder andere alternative Umsetzung am Beispiel der Klasse `TagController` ansehen.

12.3.1 Browse

Beispiel

```php
<?php

namespace Controllers;

use Entities\Article;

class IndexController extends AbstractBase
{
    public function indexAction()
    {
        $em = $this->getEntityManager();
        $query = $em
            ->createQueryBuilder()
            ->select('a, t, u')
            ->from('Entities\Article', 'a')
            ->leftJoin('a.tags', 't')
            ->leftJoin('a.user', 'u')
            ->orderBy('a.createdAt', 'DESC')
            ->getQuery()
        ;
        $articles = $query->getResult();

        $this->addContext('articles', $articles);
    }
}
```

src/Controllers/IndexController.php (Version 1)

Codebeispiel 102 *src/Controllers/IndexController.php (Version 1)*

Die Methode `IndexController#indexAction()` sollte nach den Aufgaben der letzten Lektionen ungefähr den Stand dieses Beispiels haben. Es werden alle Datensätze der Tabelle `articles` ausgelesen und im Template angezeigt. Als kleine Neuerung habe ich jedoch eine Sortierung nach dem Erstellungsdatum

ergänzt. Beachten Sie, dass hierbei natürlich der Name des Attributs in lowerCamelCase angegeben werden muss.

Unsere Aktion `browse` listet alle Datensätze einer spezifischen Datenbank-Tabelle auf. Dies muss jedoch weder zwingend in der Methode `IndexController#indexAction()` geschehen, noch müsste es eine Übersicht über alle Articles sein. In einem Newsticker ist es jedoch am sinnvollsten, eine Liste der Articles und nicht der Tags oder User auf der Startseite anzuzeigen.

Beispiel

```
<?php foreach ($articles as $article): ?>
    <article>
        <header>
            <h1><?php echo $article->getTitle(); ?></h1>
        </header>

        <div class="teaser">
            <?php echo $article->getTeaser(); ?>
        </div>

        <div class="links">
            [ <a
                href="index.php?action=read&amp;id=<?php echo
            >Details</a> ]
            [ <a
                href="index.php?action=edit&amp;id=<?php echo
            >Edit</a> ]
            [ <a
                href="index.php?action=delete&amp;id=<?php e
            >Delete</a> ]
        </div>

        <footer>
            Erstellt am
            <time datetime="<?php echo $article->getCreatedA
                <?php echo $article->getCreatedAt()->format(
            </time>
            von <?php echo $article->getUser(); ?>
            <br />
            Tags:
            <?php echo implode(' ', $article->getTags()->toA
        </footer>
    </article>
    <hr />
<?php endforeach; ?>
```

templates/IndexController/indexAction.tpl.php

Codebeispiel 103 *templates/IndexController/indexAction.tpl.php*

Dem Template sind kleinere Anpassungen widerfahren, da als Vorbereitung für die kommenden Aktionen ein paar Links ergänzt wurden. Beachten Sie bitte, dass die Angabe des Controller-Parameters in diesen Links nur unnötig ist, sofern es sich um Aktionen im `IndexController` handelt.

12.3.2 Read

Die Aktion `read` dient zur Anzeige eines bestimmten Datensatzes. In vielen Fällen handelt es sich hierbei um eine sogenannte Detailseite. Im Falle unserer Articles geht es hierbei also hauptsächlich um die Ausgabe des bisher fehlenden Attributs `$news`.

Beispiel

```php
<?php

namespace Controllers;

use Entities\Article;

class IndexController extends AbstractBase
{
    // gekuerztes Beispiel

    public function readAction()
    {
        $em = $this->getEntityManager();
        $article = $em
            ->getRepository('Entities\Article')
            ->find($_GET['id'])
        ;

        $this->addContext('article', $article);
    }
}
```

src/Controllers/IndexController.php (Version 2)

Codebeispiel 104 *src/Controllers/IndexController.php (Version 2)*

Wie schon an der Verlinkung in der *indexAction.tpl.php* ersichtlich, wird die

Detailseite von der Aktion `read` und somit der Methode `IndexController#readAction()` angezeigt, die einfach nur den passenden Datensatz anhand der übergebenen ID ermittelt. Ich habe mich der Einfachheit halber für die Nutzung einer Repository-Methode entschieden. Die Verwendung eines JOINs könnte aber durchaus sinnvoll sein, wenn die Detailseite sehr oft betrachtet wird.

Beispiel

```
<article>
    <header>
        <h1><?php echo $article->getTitle(); ?></h1>
    </header>

    <div class="text">
        <?php echo nl2br($article->getNews()); ?>
    </div>

    <div class="links">
        [ <a
            href="index.php?action=edit&amp;id=<?php echo $a
        >Edit</a> ]
        [ <a
            href="index.php?action=delete&amp;id=<?php echo
        >Delete</a> ]
    </div>

    <footer>
        Erstellt am
        <time datetime="<?php echo $article->getCreatedAt()->
            <?php echo $article->getCreatedAt()->format('d.m
        </time>
        von <?php echo $article->getUser(); ?>
        <br />
        Tags:
        <?php foreach ($article->getTags() as $tag): ?>
            <a
                href="index.php?controller=tag&amp;action=re
            ><?php echo $tag; ?></a>
        <?php endforeach; ?>
    </footer>
</article>
```

templates/IndexController/readAction.tpl.php

Codebeispiel 105 *templates/IndexController/readAction.tpl.php*

Der Inhalt des Templates ist sehr ähnlich zur *indexAction.tpl.php*. Wir benötigen in diesem Fall natürlich keine `foreach`-Schleife zur Ausgabe des `Article`, da wir nur einen Datensatz aus der Datenbank ausgelesen haben. Außerdem zeigen wir in Zeile 7 anstatt des Teasers den ausführlichen Text an und benötigen auch keinen Details-Link mehr. Wir benutzen bei der Ausgabe `nl2br()`, um mehrere Textabsätze zu ermöglichen.

Im Template wurden zudem Links für eine Aktion ergänzt, mit der man sich alle Articles eines bestimmten Tags ansehen kann. Damit dies möglich ist, verwenden wir für die Tag-Anzeige kein `implode()` mehr, sondern eine `foreach`-Schleife. Wie Sie bestimmt schon bemerkt haben, wird für die eigentliche Anzeige dieser Tag-Detailseite die Methode `TagController#readAction()` benötigt, welche wir allerdings erst später erstellen werden.

12.3.3 Edit

Kommen wir nun zum Bearbeiten eines `Article`. Dies soll mit der Aktion `edit` ermöglicht werden. Die Vorgehensweise ist zunächst einmal weitgehend identisch zur Methode `IndexController#readAction()`.

Beispiel

```
 1 <?php
 2
 3 namespace Controllers;
 4
 5 use Entities\Article;
 6
 7 class IndexController extends AbstractBase
 8 {
 9     // gekuerztes Beispiel
10
11     public function editAction()
12     {
13         $em = $this->getEntityManager();
14         $article = $em
15             ->getRepository('Entities\Article')
16             ->find($_REQUEST['id'])
17         ;
18
19         $this->addContext('article', $article);
```

```
20         }
21     }
```

src/Controllers/IndexController.php (Version 3)

Codebeispiel 106 *src/Controllers/IndexController.php (Version 3)*

Auch hier wird der gewünschte Datensatz anhand seiner ID ermittelt, allerdings müssen die Werte der Attribute nun in einem Formular angezeigt und von diesem entgegengenommen werden. Wird das Formular abgeschickt, so erhalten wir die bearbeiteten Daten über die Superglobale `$_POST`. Wir bekommen die ID des Datensatzes also zuerst aus `$_GET` und danach aus `$_POST`. Um uns hierfür eine Fallunterscheidung zu ersparen, verwenden wir `$_REQUEST['id']`.

Beispiel

```
 1 <form action="index.php?action=<?php echo $action; ?>" metho
 2
 3     <input
 4         name="id" type="hidden"
 5         value="<?php echo $article->getId(); ?>"
 6     />
 7
 8     <label for="title">Title*</label>
 9     <input
10         name="title" id="title" type="text" maxlength="80"
11         value="<?php echo $article->getTitle(); ?>"
12     />
13
14     <!-- Tagging -->
15
16     <label for="teaser">Teaser*</label>
17     <input
18         name="teaser" id="teaser" type="text" maxlength="255
19         value="<?php echo $article->getTeaser(); ?>"
20     />
21
22     <label for="news">News*</label>
23     <textarea
24         name="news" id="news"
25     ><?php echo $article->getNews(); ?></textarea>
26
27     <label for="publish_at">PublishAt</label>
28     <input
29         name="publish_at" id="publish_at" type="text"
30         value="<?php echo $article->getPublishAt()->format('
```

```
31      />
32
33      <input type="submit" class="button" value="Abschicken" /
34
35  </form>
```

templates/IndexController/editAction.tpl.php (Version 1)

Codebeispiel 107 *templates/IndexController/editAction.tpl.php (Version 1)*

In Zeile 1 sorgen wir dafür, dass die Formulareingaben an die **aktuelle Aktion** des Controllers verschickt werden. Diese Aktion ist nun sowohl für die Anzeige des Formulars als auch die Verarbeitung der Formulardaten zuständig. In Zeile 3-6 wird über ein verstecktes Formularfeld die ID des bearbeiteten Datensatzes zu den Formulardaten hinzugefügt. Ohne diese ID wäre es nicht möglich, den korrekten Datensatz in der Datenbank zu aktualisieren. Im restlichen Code des Templates sorgen wir dafür, dass alle Formularfelder auf Basis des jeweiligen Attributs mit einem Standard-Wert versehen werden.

persist und flush

Doch wie bekommen wir denn nun die bearbeiteten Formulardaten in unseren Datensatz in der Datenbank?

Beispiel

```
 1  <?php
 2
 3  namespace Controllers;
 4
 5  use Entities\Article;
 6  use Webmasters\Doctrine\ORM\Util;
 7
 8  class IndexController extends AbstractBase
 9  {
10      // gekuerztes Beispiel
11
12      public function editAction()
13      {
14          $em = $this->getEntityManager();
15          $article = $em
16              ->getRepository('Entities\Article')
17              ->find($_REQUEST['id'])
18          ;
19
```

```
20          if ($_POST) {
21              Util\ArrayMapper::setEntity($article)->setData($_
22
23              $em->persist($article);
24              $em->flush();
25
26              $this->setMessage('Article wurde aktualisiert.')
27              $this->redirect();
28          }
29
30          $this->addContext('article', $article);
31      }
32  }
```

src/Controllers/IndexController.php (Version 3b)

Codebeispiel 108 *src/Controllers/IndexController.php (Version 3b)*

Zunächst einmal benötigen wir definitiv eine `if`-Abfrage, da unsere Aktion nun unter bestimmten Umständen eine zusätzliche Aufgabe hat. Folgendes soll fortan geschehen:

1. Die Anzeige eines Formulars, das die bisherigen Daten enthält. Die ID wird hierbei im Link und somit in `$_GET` übergeben.
2. Die Aktualisierung des Datensatzes in der Datenbank, sofern das Formular abgeschickt wurde. Die ID wird in diesem Fall in einem versteckten Formularfeld und somit in `$_POST` übergeben.

Die Unterscheidung zwischen diesen beiden Möglichkeiten ist extrem simpel, wir müssen lediglich in Zeile 20 mit einer `if`-Abfrage prüfen, ob Formulardaten existieren. Die extrem kurze Bedingung ist hierbei möglich, da wir uns die automatische Typumwandlung von PHP zunutze machen (ein nicht leeres Array entspricht immer `true`). Die `if`-Abfrage dient jedoch nicht der Validierung der Eingaben, da auch für ein leeres Texteingabefeld ein Schlüssel in `$_POST` existiert.

Liegen Formulardaten vor, so wurde das Formular abgeschickt und wir können in Zeile 21 das Objekt `$article` mit den aktuellen Daten versehen. Dies geschieht mit dem `ArrayMapper`, wobei sich der Aufruf von der Variante im Konstruktor nur durch die geänderte Objekt-Variable (hier `$article`) unterscheidet. In den Zeilen 23-24 verwenden wir die gleiche Vorgehensweise wie bei neuen Datensätzen (`INSERT`) zum Speichern des aktualisierten

Datensatzes (`UPDATE`). Abschließend erstellen wir in Zeile 26 eine Flash-Notice und machen dann in Zeile 27 eine Header-Umleitung auf unsere Startseite.

> Wenn Sie Daten auf Basis von Formulareingaben speichern oder gespeicherte Daten verändern (z. B. in der Datenbank), sollten Sie die Erfolgsmeldung erst nach einer Header-Umleitung anzeigen. Versuchen Sie möglichst immer, folgenden Ablauf einzuhalten: Daten in einem Formular eingeben, per Post-Methode an den Controller übermitteln, Verarbeitung in einer Controller-Aktion, dortige Initiierung der Datenspeicherung und abschließend eine Umleitung zu einer anderen Controller-Aktion inklusive Anzeige der Erfolgsmeldung (Stichwort ***Flash-Notices***). Bei dieser Vorgehensweise handelt es sich um das sogenannte ***PRG-Entwurfsmuster*** (***Post/Redirect/Get***). Das PRG-Entwurfsmuster verhindert eine doppelte Datenspeicherung bei einem Browser-Reload.

> Für Sie als Programmierer/-in unterscheidet sich das Hinzufügen eines neuen Datensatzes nicht vom Speichern eines veränderten Datensatzes. In beiden Fällen kommen die Methoden `EntityManager#persist()` und `EntityManager#flush()` zum Einsatz. Unterschiedlich ist lediglich die Art und Weise, wie Sie zuvor die Variable mit dem Objekt erhalten haben.

12.3.4 Add

Beim Hinzufügen eines neuen Datensatzes benötigen wir in der Controller-Aktion `add` wieder eine Unterscheidung zwischen zwei Möglichkeiten. Doch zunächst müssen wir ein paar kleine Vorbereitungen treffen.

Beispiel

```
1 <ul id="navi">
2     <li><a href="index.php">Home</a></li>
3     <li><a href="index.php?controller=tag&amp;action=add">Crea
4     <li><a href="index.php?action=add">Create Article</a></li
5 </ul>
```

templates/navi.tpl.php

Codebeispiel 109 *templates/navi.tpl.php*

Damit haben wir endlich eine funktionierende Navigation, mit der wir zum Formular für das Anlegen eines `Article` gelangen können.

Kommen wir nun zu den beiden angekündigten Fällen:

1. Anzeige des (leeren) Formulars
2. Speicherung des neuen Datensatzes in der Datenbank anhand der Formulardaten.

Beispiel

```php
<?php

namespace Controllers;

use Entities\Article;
use Webmasters\Doctrine\ORM\Util;

class IndexController extends AbstractBase
{
    // gekuerztes Beispiel

    public function addAction()
    {
        $em = $this->getEntityManager();
        $article = new Article();

        if ($_POST) {
            Util\ArrayMapper::setEntity($article)->setData($_

            $article->setUser(
                $em->getRepository('Entities\User')->find(1)
            );

            $em->persist($article);
            $em->flush();

            $this->setMessage('Article wurde gespeichert.');
            $this->redirect();
        }

        $this->addContext('article', $article);
        $this->setTemplate('editAction');
    }
```

```
34 }
```

src/Controllers/IndexController.php (Version 4)

Codebeispiel 110 *src/Controllers/IndexController.php (Version 4)*

Der Unterschied zwischen den Aktionen **edit** und **add** ist lediglich, welche Daten vor der Befüllung mit den Formulardaten im Objekt vorliegen. In Zeile 15 wurde nämlich zunächst eine leere Instanz der Klasse `Article` erzeugt und diese in der Variablen `$article` abgelegt. Wir haben also beim Editieren und Anlegen **immer** eine Instanz vorliegen, auf die unser Formular-Template zugreifen kann. Hierdurch vermeiden wir eine ganze Reihe **unnötiger** `if`-Abfragen in den `value`-Attributen unserer Formularfelder.

Der Einfachheit halber wurde für die Entity-Klasse in Zeile 5 ein Kurzname angelegt, denn wir wollen in den einzelnen Methoden nicht immer an die Angabe des Namespaces `Entities` denken müssen. Das `Article`-Objekt wird dann in Zeile 18 mit den Formulardaten befüllt, und in den Zeilen 20-22 benutzen wir die Methode `EntityRepository#find()` mit dem FK `1`, um zusätzlich den Ersteller des Datensatzes zu hinterlegen. Sobald wir einen funktionierenden Login haben, müssen wir natürlich an dieser Stelle die korrekte ID des eingeloggten Benutzers verwenden.

Das Template zum Anlegen und Editieren eines Datensatzes sind nahezu identisch, deswegen sorgen wir in Zeile 32 für eine Wiederverwendung und vermeiden so eine **unnötige** Code-Duplizierung. Dies ist allerdings nur möglich, weil wir in Zeile 1 des Templates die Aktion als Variable hinterlegt haben.

Wichtig für eine Template-Wiederverwendung ist, dass die Controller-Aktion beim Anlegen und Bearbeiten eines Datensatzes **immer** für eine Instanz der entsprechenden Doctrine-Entity sorgt und dieses Objekt mittels `AbstractBase#addContext()` dem Template zur Verfügung stellt, damit dort die Getter-Aufrufe nicht scheitern.

Falls sich die Formular-Anzeige in beiden Fällen leicht unterscheiden soll, so ist trotzdem eine Template-Wiederverwendung möglich. Sie müssen das Formular lediglich mit einer `if-else`-Anweisung ergänzen, in der Bedingung den Inhalt von `$action` überprüfen und die gesamte Anweisung an der Stelle im Template positionieren, wo eine unterschiedliche Ausgabe benötigt wird.

Eine kleine Wiederholung

Schauen wir uns das Template noch einmal genauer an und konzentrieren uns hierbei auf das Formularfeld `publish_at`.

Beispiel

```
<label for="publish_at">PublishAt</label>
<input
    name="publish_at" id="publish_at" type="text"
    value="<?php echo $article->getPublishAt()->format('Y-m-d H
/>
```

templates/IndexController/editAction.tpl.php (Ausschnitt)

Codebeispiel 111 *templates/IndexController/editAction.tpl.php (Ausschnitt)*

Beim Editieren eines bestehenden Datensatzes ist der Aufruf der `format`-Methode eigentlich unproblematisch, da wir zumindest theoretisch nur Datensätze mit gültigen Datumswerten speichern und wir somit immer ein Objekt haben, auf dem wir die Methode aufrufen können. Praktisch fehlt uns diese Eingabevalidierung aber noch (dazu kommen wir in Lektion 13). Auf jeden Fall wird das Formularfeld mit dem Datum aus dem Datensatz vorbelegt (Standard-Wert des Formularfeldes).

Doch wie sieht es beim Anlegen eines neuen Datensatzes bzw. beim ersten Aufruf des Anlegen-Formulars aus? Auch hier werden die Formularfelder über Getter-Aufrufe vorbelegt. Hat das entsprechende Attribut einen Default-Wert, so wird dieser angezeigt. Ansonsten ist das Formularfeld initial leer, sofern der Getter nicht etwas anderes festlegt. Das Attribut `$publishAt` hat keinen Default-Wert (entspricht dem Wert `null`). Der Getter `getPublishAt` liefert deshalb ein `new Webmasters\Doctrine\ORM\Util\DateTime(null)` zurück. Würden wir die normale `DateTime`-Klasse von PHP verwenden, so würden wir beim Wert `null` den aktuellen Zeitpunkt erhalten und somit auch beim ersten Formularaufruf angezeigt bekommen. Ein bereits teilweise ausgefülltes Formular ist jedoch für die meisten Anwender von der Usability her verwirrend und für uns Entwickler zudem schwerer zu validieren. Wir verwenden deswegen nicht die Original-Klasse und erhalten so beim Aufruf der `format`-Methode

einfach eine leere Datumsausgabe bzw. ein nicht vorbelegtes Formularfeld. Dies ist bei einer engen Kopplung von Formular und `Article`-Objekt, wie wir sie verwenden, von der Usability her stimmiger.

12.3.5 Delete

Zum Löschen eines Datensatzes benötigen wir etwas Neues, nämlich die Methode `EntityManager#remove()`. Diese Methode ermöglicht eine objektorientierte Nutzung des SQL-Befehls DELETE. Sie müssen also, wie bei Doctrine üblich, selbst keinen SQL-Code schreiben.

Beispiel

```php
<?php

namespace Controllers;

use Entities\Article;
use Webmasters\Doctrine\ORM\Util;

class IndexController extends AbstractBase
{
    // gekuerztes Beispiel

    public function deleteAction()
    {
        $em = $this->getEntityManager();
        $article = $em
            ->getRepository('Entities\Article')
            ->find($_GET['id'])
        ;

        $em->remove($article);
        $em->flush();

        $this->setMessage('Article wurde entfernt.');
        $this->redirect();
    }
}
```

src/Controllers/IndexController.php (Version 5)

Codebeispiel 112 *src/Controllers/IndexController.php (Version 5)*

Ähnlich wie bei **Read** und **Edit** wird in der Aktion `delete` zunächst der passende

Datensatz anhand der übergebenen ID ermittelt. Danach teilen wir Doctrine in Zeile 20 mit, dass wir diesen Datensatz löschen möchten. Wir müssen den Datensatz also erst einmal als Objekt vorliegen haben, um ihn dann löschen zu können. Wie auch beim Speichern wird dies übrigens erst beim Flush ausgeführt. Dies mag insgesamt etwas umständlich erscheinen, ist aber zumindest konsistent zu unserer bisherigen Vorgehensweise beim Editieren. Wir benötigen übrigens (noch) kein Template für die Aktion `delete`, da wir diese Aktion im Moment **immer** mit einer Header-Umleitung zur Startseite beenden.

12.4 Tagging

Das Template *editAction.tpl.php* hat aktuell ein Problem, und zwar an der Stelle, wo derzeit noch der HTML-Kommentar als Platzhalter steht. An dieser Stelle fehlt nämlich noch das Tagging, d.h. die Auswahl der gewünschten Tags, was beispielsweise über ein `select`-Tag geschehen kann.

Beispiel

```
 1 <label for="tags">Tagging*</label>
 2 <select name="tag_ids[]" id="tags" multiple="multiple">
 3     <?php foreach ($tags as $tag): ?>
 4         <option
 5             value="<?php echo $tag->getId(); ?>"
 6             <?php if ($article->hasTag($tag)): ?>
 7                 selected="selected"
 8             <?php endif; ?>
 9         ><?php echo $tag; ?></option>
10     <?php endforeach; ?>
11 </select>
```

templates/IndexController/editAction.tpl.php (Version 2 - Ausschnitt Auswahl Tags)

Codebeispiel 113 *templates/IndexController/editAction.tpl.php (Version 2 - Ausschnitt Auswahl Tags)*

In den Zeilen 3-10 wird über eine `foreach`-Schleife für jeden Eintrag des Arrays `$tags` eine Option erzeugt. In den Zeilen 6-8 füllen wir dann sozusagen den bisherigen Wert des Attributs in das Formular-Feld. Da das `select`-Tag kein `value`-Attribut hat, muss man hierfür zu einer etwas komplexeren Lösung

greifen. Mit einer `if`-Abfrage sorgen wir für das Hinzufügen des `selected`-Attributs zur aktuellen Option des Schleifen-Durchlaufs, sofern das Objekt in `$tag` bereits in der entsprechenden `ArrayCollection` von `$article` enthalten ist.

Beispiel

```php
<?php

namespace Controllers;

use Entities\Article;
use Webmasters\Doctrine\ORM\Util;

class IndexController extends AbstractBase
{
    // gekuerztes Beispiel

    public function editAction()
    {
        $em = $this->getEntityManager();
        $article = $em
            ->getRepository('Entities\Article')
            ->find($_REQUEST['id'])
        ;
        $tags = $em
            ->getRepository('Entities\Tag')
            ->findAll()
        ;

        if ($_POST) {
            Util\ArrayMapper::setEntity($article)->setData($

            $article->clearTags();

            $tag_ids = isset($_POST['tag_ids']) ? $_POST['ta
            foreach ($tag_ids as $id) {
                $article->addTag(
                    $em->getRepository('Entities\Tag')->find
                );
            }

            $em->persist($article);
            $em->flush();

            $this->setMessage('Article wurde aktualisiert.')
            $this->redirect();
```

```
41          }
42
43          $this->addContext('article', $article);
44          $this->addContext('tags', $tags);
45      }
46
47      // gekuerztes Beispiel
48  }
```

src/Controllers/IndexController.php (Version 6)

Codebeispiel 114 *src/Controllers/IndexController.php (Version 6)*

Die Aktion zum Editieren eines `Article` benötigt für das Tagging zwei Ergänzungen. Das Template setzt nämlich in der `foreach`-Schleife eine Variable `$tags` mit einem Array aller Tags voraus, welches wir deshalb in den Zeile 19-22 des Controllers ermitteln und in Zeile 44 dem Template zur Verfügung stellen.

Doch wozu dienen die Zeilen 27-34 des Controllers? Zwischen `Article` und `Tag` besteht ja bekanntermaßen eine n:m-Beziehung, womit jeder Article mehrere Tags haben kann. Und jedes Tag kann mehrere Articles haben, was hier aber nicht wirklich relevant ist. Das Attribut `$tags` in `Article` muss somit eine `ArrayCollection` beinhalten. In Zeile 27 löschen wir zunächst alle bisherigen Tags aus dieser Sammlung. In Zeile 29 prüfen wir dann, ob der Schlüssel `tag_ids` im Array `$_POST` enthalten ist. Ist dies nicht der Fall, so soll die Variable `$tag_ids` mit einem leeren Array befüllt werden. Dies wäre übrigens der Fall, wenn im Formular gar kein `Tag` ausgewählt wurde.

Moment! Es gibt die Möglichkeit, dass gar kein `Tag` ausgewählt wurde? Ist in einem Select nicht immer die erste Option standardmäßig ausgewählt, sofern der Anwender keine Auswahl getroffen hat? Nein, sofern man (wie wir) einen Select mit Mehrfachauswahl verwendet, kann genau dieser Fall eintreten. Man kann nämlich die Auswahl mehrerer Optionen ermöglichen, indem man die Angabe `multiple="multiple"` im öffnenden `select`-Tag ergänzt.

Und wieso benötigen wir dann in Zeile 30-34 eine `foreach`-Schleife, die dieses befüllte oder leere Array in `$tag_ids` durchläuft? Unser Formular-Feld soll mehrere Fremdschlüssel-IDs (FKs) übermitteln können. Die Übertragung ist somit nur als **Array** möglich und wird durch die Schreibweise `tag_ids[]` im Template ermöglicht. Die Schreibweise mit oder ohne eckige Klammern

bestimmt also, ob Sie den Datentyp Array oder String von einem Formularfeld erhalten. Zusammen mit dem `multiple`-Attribut sorgt dies dafür, dass das Formularfeld als Array mit **mehreren** Werten übertragen werden kann. Die FKs im Array durchlaufen wir dann in Zeile 30-34 einzeln und fügen in Zeile 31-33 das passende `Tag`-Objekt für jede ID zum `Article`-Objekt hinzu.

Würden wir mit dem Select beispielsweise einen einzelnen User anstatt mehrerer Tags auswählen, so wären die eckigen Klammern und das `multiple="multiple"` überflüssig, da wir nur einen einzigen Wert für das Formularfeld übertragen müssten. Wir könnten dann in der Controller-Aktion ähnlich wie in der Methode `IndexController#addAction()` beim Aufruf der Methode `Article#setUser()` ohne `foreach`-Schleife und ohne den Delegator für `ArrayCollection#clear()` arbeiten, da das Attribut `$user` keine `ArrayCollection` beherbergt.

12.5 Zusammenfassung

Glückwunsch! Damit haben Sie das Pflichtprogramm hinter sich gebracht. Doch arbeiten Sie bitte trotzdem diesen Kurs weiter durch, denn ohne die folgenden fortgeschrittenen Themen sollten Sie besser keine Webanwendung mit Doctrine umsetzen. Derzeit fehlen beispielsweise noch so essenzielle Themen wie Validierung von Benutzereingaben und Sicherheit, die wir erst in den beiden folgenden Lektionen besprechen werden.

12.6 Testen Sie Ihr Wissen

1. Wofür steht das Akronym CRUD?
2. Wofür steht das Akronym BREAD?
3. Weswegen beschreibt BREAD die Benutzeroberfläche einer MVC-Anwendung besser als CRUD?
4. Wodurch unterscheiden sich die Operationen Read, Edit und Delete von

den (eher allgemeinen) Operationen Browse und Add?

12.7 Aufgaben zur Selbstkontrolle

Übung 33:

Implementieren Sie die vorgestellten Aktionen dieser Lektion inklusive der passenden Templates und testen Sie alle Aktionen im Browser.

Übung 34:

Implementieren Sie nun auch das Tagging in der Aktion `add`. Hier ist übrigens kein Aufruf der Delegator-Methode `Article#clearTags()` nötig, da wir ja mit einem leeren Objekt arbeiten und somit dank unseres Konstruktors bereits eine leere `ArrayCollection` vorliegen haben.

Übung 35:

Im Template *templates/IndexController/readAction.tpl.php* gibt es einen Link für die Methode `TagController#readAction()`. Setzen Sie diese Methode nun um. Sie soll ein Tag anhand der übergebenen ID auslesen und per Getter alle dazugehörigen Articles ausgeben wird. Beim Template ist übrigens eine Wiederverwendung von *templates/IndexController/indexAction.tpl.php* möglich, wenn man wirklich nur die Articles ausgeben will.

Übung 36:

Setzen Sie die noch fehlende Aktion zum Anlegen von Tags um. Verwenden Sie hierfür die bereits vorbereitete Methode `TagController#addAction()`. Unser Template *templates/TagController/editAction.tpl.php* benötigt nun ein Formular, in dem wir den Wert für `$title` eingeben können. Es ist hierbei

nicht nötig, die Beziehung zu unseren Articles zu berücksichtigen, da die Zuweisung bereits über die `Article`-Aktionen möglich ist.

Hinweis: Die bereits vorhandene Ausgabe der schon existierenden Tags soll als optische Hilfestellung unter dem Formular erfolgen und somit erhalten bleiben.

12.8 Optionale Aufgaben

Übung 37:

Unsere Aktion `search` funktioniert wegen der soeben erfolgten Ergänzung des Formulars im Template *templates/TagController/editAction.tpl.php* nicht mehr. Ändern Sie die Aktion folgendermaßen ab:

- Es soll nun das Attribut `$title` der Entity `Article` durchsucht werden. Vergessen Sie nicht, den Alias im Query anzupassen, damit der Code weiterhin lesbar bleibt.
- Verwenden Sie zur Anzeige der Suchergebnisse das bereits existierende Template *templates/IndexController/indexAction.tpl.php*.
- Ergänzen Sie in diesem Template über der Ausgabe der Articles ein Formular, in dem ein Suchbegriff eingegeben werden kann. Dieses Formular soll im HTML-Code im öffnenden `form`-Tag die ID `search` bekommen. Denken Sie auch daran, die korrekte Aktion für den Versand der Formulardaten anzugeben.
- Das Formular soll nur angezeigt werden, wenn die Variable `$action` den Wert `'index'` oder `'search'` hat. Ergänzen Sie eine entsprechende `if`-Abfrage im Template.
- Der Platzhalter im Query der Methode soll nicht mehr mit dem festen Wert `'%h%'` befüllt werden. Verketten Sie stattdessen die Prozentzeichen mit dem Suchbegriff.
- Ergänzen Sie im Query außerdem eine Sortierung nach dem

Erstellungsdatum. Der neueste `Article` soll zuerst angezeigt werden.

13 Fortgeschrittene Techniken

In dieser Lektion lernen Sie
• wie Sie Benutzereingaben validieren können. • wie Sie Ihren Code mit Repositories lesbarer gestalten. • wie Sie Zufallsdatensätze auslesen.

13.1Doctrine um Validierungen erweitern

Bisher haben wir unsere Entity-Objekte einfach in der Datenbank gespeichert und gehofft, dass dies erfolgreich ist. Hierbei war nicht relevant, ob ein Attribut auf eine bestimmte Weise oder überhaupt befüllt war. Auch liefen Sie vermutlich in die eine oder andere Fehlermeldung, weil der Wert eines Alternativschlüssels schon in einem anderen Datensatz existierte. All dies lässt sich mit Validierungen lösen.

Validierungen sind bei Doctrine 2 (im Gegensatz zu Doctrine 1) nicht mehr Teil der Core-Bibliotheken. Um die Umsetzung möglichst einfach und lesbar zu halten, habe ich mich gegen die Nutzung der Doctrine Lifecycle Events und für eine Umsetzung innerhalb der Webmasters Doctrine Extensions entschieden.

Beispiel

```php
<?php

namespace Validators;

use Webmasters\Doctrine\ORM\EntityValidator;

class TagValidator extends EntityValidator
{
    public function validateTitle($title)
    {
        if (empty($title)) {
            $this->addError('Das Feld Title ist leer.');
        } elseif (strlen($title) < 3) {
```

```
14              $this->addError('Der Title sollte mindestens 3 Ze
15          }
16      }
17  }
```

src/Validators/TagValidator.php (Version 1)

Codebeispiel 115 *src/Validators/TagValidator.php (Version 1)*

In diesem Beispiel haben wir eine ***Validator-Klasse*** für ein `Tag`-Objekt geschrieben. Eine verpflichtende Namenskonvention ist hierbei, dass der Dateiname des Validators mit dem Namen der Entity ohne Namespace (hier `Tag`) beginnt und mit `Validator.php` endet. In Zeile 3 nutzen wir den Namespace `Validators`, den wir in Lektion 4 für unsere Validator-Klassen festgelegt haben. Da jede Validator-Klasse einen bestimmten Umfang von Attributen und Methoden als Basis benötigt, benutzen wir eine Vererbung von der Klasse `Webmasters\Doctrine\ORM\EntityValidator`.

Beispiel

```
 1  <?php
 2
 3  namespace Webmasters\Doctrine\ORM;
 4
 5  class EntityValidator
 6  {
 7      protected $em;
 8      protected $entity;
 9      protected $errors = array();
10
11      public function __construct($em, $entity)
12      {
13          $this->em = $em;
14          $this->entity = $entity;
15
16          $this->validateData();
17      }
18
19      public function validateData()
20      {
21          $data = Util\ArrayMapper::setEntity($this->entity)->
22
23          foreach ($data as $key => $val) {
24              $validate = 'validate' . ucfirst($key);
25              if (method_exists($this, $validate)) {
26                  $this->$validate($val);
27              }
```

```
28          }
29      }
30
31      public function getEntityManager()
32      {
33          return $this->em;
34      }
35
36      public function getRepository($class = null)
37      {
38          if (empty($class)) {
39              $class = get_class($this->entity);
40          }
41
42          return $this->getEntityManager()->getRepository($cla
43      }
44
45      public function getEntity()
46      {
47          return $this->entity;
48      }
49
50      public function addError($error)
51      {
52          $this->errors[] = $error;
53      }
54
55      public function getErrors()
56      {
57          return $this->errors;
58      }
59
60      public function isValid()
61      {
62          return empty($this->errors);
63      }
64  }
```

vendor/webmasters/doctrine-extensions/lib/Webmasters/Doctrine/ORM/EntityValidator.php

Codebeispiel 116 *vendor/webmasters/doctrine-extensions/lib/Webmasters/Doctrine/ORM/EntityValidator.php*

Für den Aufruf der eigentlichen Validierungs-Methoden unserer Klasse `TagValidator` kommt ein ähnliches Prinzip wie in der Methode `setDaten()` bzw. `setData()` zum Einsatz. Hierzu wird in Zeile 21 unserer Elternklasse das Entity-Objekt mit der Methode `ArrayMapper#toArray()` in ein Array umgewandelt. Grundsätzlich können Sie also jedes Attribut validieren. Die Doctrine-Entity

muss lediglich über einen Getter für das jeweilige Attribut verfügen, sonst fehlt das Attribut nämlich im Array. Auf jeden Fall fehlt jedoch die ID im Array, da wir dies mit dem Wert `false` für den ersten Parameter von `ArrayMapper#toArray()` festgelegt haben. Auf die Validierung der ID können wir nämlich verzichten, da sie ja automatisch vergeben wird.

Das Array wird anschließend mit einer `foreach`-Schleife durchlaufen und für jedes Attribut wird nach einer passenden Methode für die Validierung gesucht, die dann wiederum aufgerufen wird. Den entsprechenden Code erben wir einfach von unserer Elternklasse, wir müssen ihn also **nicht** für jede Validator-Klasse erneut schreiben. Sie müssen lediglich in der Validator-Klasse eine Methode für jedes zu validierende Attribut erstellen und hierbei eine vergleichbare Namenskonvention wie bei unseren Gettern bzw. Settern nutzen. Anstatt mit `get` bzw. `set` beginnt der Methoden-Name allerdings mit `validate`, was in Zeile 24 der Elternklasse festgelegt wird.

Die Beispiel-Methode `TagValidator#validateTitle()` validiert den Wert des Attributs `$title`, der in Zeile 26 der Elternklasse als Parameter an die Methode übergeben wird. Für die eigentliche Validierung wird eine `if-elseif`-Anweisung benutzt. Schlägt eine der Bedingungen fehl, so wird die Methode `EntityValidator#addError()` genutzt, um die entsprechende Validierungs-Meldung in einem Fehler-Array abzulegen.

Beispiel

```
 1 <?php
 2
 3 namespace Controllers;
 4
 5 use Entities\Tag;
 6 use Webmasters\Doctrine\ORM\Util;
 7
 8 class TagController extends AbstractBase
 9 {
10     public function addAction()
11     {
12         $em = $this->getEntityManager();
13
14         $tag = new Tag();
15         $tags = $em
```

```
16              ->getRepository('Entities\Tag')
17              ->findAll()
18          ;
19
20          if ($_POST) {
21              Util\ArrayMapper::setEntity($tag)->setData($_POS
22
23              $validator = $em->getValidator($tag);
24              if ($validator->isValid()) {
25                  $em->persist($tag);
26                  $em->flush();
27
28                  $this->setMessage('Tag wurde gespeichert.');
29                  $this->redirect('add', 'tag');
30              }
31
32              $this->addContext('errors', $validator->getError
33          }
34
35          $this->addContext('tag', $tag);
36          $this->addContext('tags', $tags);
37          $this->setTemplate('editAction');
38      }
39  }
```

src/Controllers/TagController.php

Codebeispiel 117 *src/Controllers/TagController.php*

Die Nutzung der Validator-Klassen im Controller ist relativ simpel, Sie müssen lediglich den `EntityManager` mit der Methode `EntityManager#getValidator()` nach einer Instanz der Validator-Klasse für ein bestimmtes Entity-Objekt fragen und diese wie in Zeile 23 in einer Variablen ablegen. Die Methode ermittelt dann intern den richtigen Namen für die Validator-Klasse und erzeugt eine Instanz mit dem Schlüsselwort `new`. Beachten Sie bitte, dass diese Methode kein Teil des Doctrine Core ist, sondern Teil der Webmasters Doctrine Extensions. Würden wir den normalen Entity Manager von Doctrine nutzen, so stünde uns diese Methode **nicht** zur Verfügung – und Sie wissen nun, warum wir diesen nicht mehr verwenden.

Allerdings ist es auch nicht sonderlich schwierig, die Validierungen ohne den angepassten Entity Manager und dessen `EntityManager#getValidator()` zu verwenden. Man muss hierfür lediglich selbst eine Instanz der passenden Validator-Klasse in `$validator` ablegen und deren Konstruktor das Entity-Objekt und `$em` übergeben.

Beispiel

```
$validator = new \Validators\TagValidator($tag, $em);
```

manuelle Instanziierung

Codebeispiel 118 *manuelle Instanziierung*

So besteht dann auch die Möglichkeit, eine alternative Validierungsklasse zu verwenden, die nicht an die Namenskonvention unseres EntityManagers gebunden ist. Dies könnte beispielsweise nötig werden, wenn beim Anlegen und Editieren einer Entity gravierende Unterschiede in der Validierung existieren und man deswegen getrennte Klassen nutzen möchte.

Abschließend müssen wir die Ausführung des eigentlichen Speicherns, die Flash-Notice und die Header-Umleitung verhindern, sofern das Objekt keine validen Werte hat. Dies geschieht mit der `if`-Abfrage in den Zeilen 24-30 der Controller-Aktion und dem dortigen Aufruf der Methode `EntityValidator#isValid()`. In Zeile 32 fragen wir dann noch den Validator mit `EntityValidator#getErrors()` nach den Fehlermeldungen und legen diese für das Partial *templates/errors.tpl.php* in der Variablen `$errors` ab. Diese Zeile wird aufgrund der beiden `if`-Abfragen und der Header-Umleitung in Zeile 29 nur ausgeführt, sofern:

1. Formulardaten vorhanden sind.
2. die Validitätsprüfung scheitert.
3. durch die gescheiterte Prüfung auch keine Header-Umleitung ausgeführt wird.

Die Anzeige der Fehlermeldungen erfolgt dann über eine `foreach`-Schleife im Partial *errors.tpl.php* als unnummerierte Liste, da wir grundsätzlich von mehreren Validitätsproblemen ausgehen.

Controller, wir haben ein Problem

Schauen wir uns den Code in `TagController#addAction` noch einmal etwas genauer an und gehen wir von der Eingabe eines Titels mit lediglich zwei Buchstaben aus. Was passiert?

In Zeile 14 instanziieren wir zunächst ein leeres `Tag`-Objekt. Das Formular wurde abgeschickt, somit ist die Bedingung in Zeile 20 `true`. In Zeile 21 wird dann unser Objekt mit den Daten aus dem Formular befüllt. Das Objekt enthält somit nun die **nicht validen** Daten (Titel zu kurz). Die Bedingung in Zeile 24 ist dementsprechend `false` und das Objekt wird nicht gespeichert. Allerdings werden in Zeile 32, 35 und 36 dem Partial *templates/errors.tpl.php* und dem Template *editAction.tpl.php* Daten zur Verfügung gestellt. Im Falle von Zeile 35 ist dies unser nicht valides Objekt.

Die Header-Umleitung in Zeile 29 wurde nicht ausgeführt, wir sehen somit nun im Browser wieder das Formular. Genauso wie beim Bearbeiten eines Datensatzes werden die Formularfelder hierbei mit den Werten aus dem Objekt vorbelegt. Auch in diesem Fall greifen wir also mittels der Getter auf unser Objekt zu. Hier sehen wir dann auch gleich einen weiteren Vorteil der Wiederverwendung von *editAction.tpl.php*. Egal, ob wir einen neuen Datensatz anlegen wollen, die aktuellen Eingaben bei einem Validierungsproblem erneut anzeigen wollen oder beim Bearbeiten die Werte aus der Datenbank anzeigen wollen, alles geht ohne **unnötige** Code-Duplizierung mit dem gleichen Template. Wir müssen lediglich für eine korrekte Befüllung des Objekts mit den aktuellen Daten sorgen.

Übung 38:

1. Implementieren Sie den Validator *src/Validators/TagValidator.php* in Ihrem Projekt und passen Sie hierfür Ihren `TagController` wie dargestellt an.
2. Testen Sie die Änderungen, indem Sie beispielsweise ein Tag mit leerem oder zu kurzem Wert für `$title` befüllen. Eine Speicherung sollte in beiden Fällen nicht mehr klappen. Stattdessen sollte eine passende Fehlermeldung ausgegeben werden.

13.1.1 Validierungsbedingungen für Datumswerte

Die Klasse `Webmasters\Doctrine\ORM\Util\DateTime` bietet zwei Methoden, die Ihnen bei der Validierung von Datumseingaben helfen können.

Beispiel

```
$date->isValid()
$date1->isValidClosingDate($date2)
```

Validierungsbedingungen für Datumswerte

Codebeispiel 119 *Validierungsbedingungen für Datumswerte*

Die erste Methode können Sie bereits problemlos für die Validierung des Attributs `$publishAt` in der Entity `Article` einsetzen, da Sie hierfür lediglich wissen müssen, in welcher Variable das `DateTime`-Objekt vorliegt. Bei der zweiten Methode sieht die Sache schon etwas anders aus, da unsere Validator-Methode immer nur einen Parameter übergeben bekommt und wir hier zwei Datumswerte `$date1` und `$date2` benötigen.[12]

---

12 Wenn `$date1` vor `$date2` liegt, so erhalten wir ein `true` als Rückgabewert von der Methode.

Beispiel

```
 1 <?php
 2
 3 namespace Validators;
 4
 5 use Webmasters\Doctrine\ORM\EntityValidator;
 6 use Webmasters\Doctrine\ORM\Util;
 7
 8 class ArticleValidator extends EntityValidator
 9 {
10     public function validatePublishAt($publishAt)
11     {
12         $now = new Util\DateTime('now');
13
14         if (!$publishAt->isValid()) {
15             $this->addError('Das Feld PublishAt muss einen k
16         } elseif (!$now->isValidClosingDate($publishAt)) {
17             $this->addError('Das Feld PublishAt darf kein Da
18         }
19     }
20 }
```

src/Validators/ArticleValidator.php

Codebeispiel 120 *src/Validators/ArticleValidator.php*

Zunächst erstellen wir in Zeile 12 ein `DateTime`-Objekt für den aktuellen

Zeitpunkt. Wir benutzen hierfür die Klasse `Webmasters\Doctrine\ORM\Util\DateTime`, da nur diese die Methode `DateTime#isValidClosingDate()` hat.[13] Danach überprüfen wir in Zeile 14, ob ein gültiges Datum eingegeben wurde. Wurde gar kein oder ein ungültiges Datum eingegeben, so wird eine Fehlermeldung ausgegeben. In Zeile 16 überprüfen wir dann, ob das eingegebene Datum in der Vergangenheit liegt. Ist dies der Fall, so wird ebenfalls eine Fehlermeldung ausgegeben. Achten Sie in beiden Fällen darauf, die Bedingung mit einem Ausrufezeichen zu invertieren.

13 Auch die Methode `DateTime#isValid()` ist nicht in der standardmäßigen Klasse `DateTime` vorhanden.

Achtung

Die Methode `DateTime#isValidClosingDate()` überprüft lediglich die **Differenz in Tagen** zwischen den beiden Datumswerten mittels `DateTime#diff()`. Ein Zeitpunkt wird also bei obiger Validierung **nicht** als in der Vergangenheit liegend erachtet, sofern er das gleiche Datum, aber eine frühere Uhrzeit beinhaltet.[14]

14 Dies wäre durch eine sekundengenaue Prüfung lösbar. Allerdings würde diese in unserem Fall vermutlich Probleme verursachen, sobald wir die Usability unserer Anwendung verbessern; dazu gleich mehr.

Übung 39:

1. Implementieren Sie den Validator *src/Validators/ArticleValidator.php* in Ihrem Projekt. Passen Sie hierzu auch die Aktionen `add` und `edit` im `IndexController` an.
2. Testen Sie die Änderungen, indem Sie einen leeren Wert bzw. ein Datum in der Vergangenheit für das Attribut `$publishAt` benutzen.

Verbessern wir nun noch die Usability unserer Anwendung, indem wir bei einem leeren Formularfeld den aktuellen Zeitpunkt als Wert für `$publishAt` verwenden.

Beispiel

```php
<?php

namespace Entities;

use Doctrine\ORM\Mapping as ORM;
use Gedmo\Mapping\Annotation as Gedmo;
use Webmasters\Doctrine\ORM\Util;

/**
 * @ORM\Entity
 * @ORM\Table(name="articles")
 */
class Article
{
    // gekuerztes Beispiel

    /**
     * @ORM\Column(name="publish_at", type="datetime")
     */
    protected $publishAt;

    // gekuerztes Beispiel

    public function setPublishAt($publishAt)
    {
        if (empty($publishAt)) {
            $publishAt = 'now';
        }

        $this->publishAt = new Util\DateTime($publishAt);
    }

    // gekuerztes Beispiel
}
```

src/Entities/Article.php

Codebeispiel 121 *src/Entities/Article.php*

Ein leerer String als Parameter für den Konstruktor wird bei unserer Klasse `Webmasters\Doctrine\ORM\Util\DateTime` akzeptiert, und im Gegensatz zur Original-Klasse wird hierdurch kein Objekt für den aktuellen Zeitpunkt erzeugt. Die Methode `DateTime#isValid()` betrachtet das entsprechende Objekt jedoch als **nicht valide** und unser `ArticleValidator` würde deshalb bei einem leeren Formularfeld eine Speicherung des Datensatzes verhindern. Das Eingabefeld soll jedoch optional sein, was man an dem fehlenden Stern im Formular

erkennen kann. Dieses Problem lässt sich relativ einfach lösen, indem wir im Fall eines leeren Strings den String `'now'` als Parameter für `DateTime` nutzen.[15] Da der Wert für das Attribut `$publishAt` somit schon beim Befüllen des Objekts mit `ArrayMapper#setData()` ermittelt wird und der Wert von `$now` erst etwas später in der Methode `ArticleValidator#validatePublishAt()`, liegt der erste Wert zeitlich kurz vor dem zweiten Wert. Die Methode `DateTime#isValidClosingDate()` könnte somit bei einem sekundengenauen Vergleich zumindest theoretisch an diesem Umstand scheitern.

---

15 Sofern sich für ein Formularfeld ein Schlüssel im Array `$_POST` befindet, wird durch `ArrayMapper#setData()` auch immer dessen Setter aufgerufen.

13.1.2 Öffentliche Methoden von EntityValidator

Beispiel

```
$entity = $this->getEntity();
$em = $this->getEntityManager();
$repository = $this->getRepository('Entities\Article');
$repository = $this->getRepository(); // Parameter ist optional
```

Öffentliche Methoden von EntityValidator

Codebeispiel 122 *Öffentliche Methoden von EntityValidator*

Mit Hilfe der drei Methoden `EntityValidator#getEntity()`, `EntityValidator#getEntityManager()` und `EntityValidator#getRepository()` ist praktisch jede Art von Validierung möglich, die Zugriff auf das komplette Objekt oder sogar auf die Datenbank benötigt. Beispielsweise könnten wir Dank `EntityValidator#getEntity()` mit `DateTime#isValidClosingDate` problemlos zwei Datumseingaben überprüfen, da wir mittels Getter ja Zugriff auf den zweiten Wert haben.

Die Angabe des Parameters ist übrigens bei der Methode `EntityValidator#getRepository()` optional. Wird kein Wert angegeben, so wird dieser aus dem zu validierenden Objekt ermittelt. Wird also beispielsweise ein `Tag`-Objekt validiert, so wird auch ein `Tag`-Repository benutzt. Dies ist jedoch eine Besonderheit der Methode in den Validator-Klassen.

13.2 Datenbankabfragen in Repositories auslagern

Wenn wir mit dem `QueryBuilder` eine Datenbankabfrage formulieren und dabei Wert auf eine gute Lesbarkeit legen, so erzeugen wir durch die manuellen Zeilenumbrüche relativ viele Zeilen Code. Diese Zeilen blähen unsere Controller-Klassen unnötig auf und sind zudem nicht wiederverwendbar. Betrachten wir beispielsweise einmal eine Datenbankabfrage, die ermittelt, ob es zu einem bestimmten Tag `$tag` noch andere Datensätze mit identischem Title gibt.

Beispiel

```
$query = $em
    ->createQueryBuilder()
    ->select('t')
    ->from('Entities\Tag', 't')
    ->where('t.title = :title')
    ->andWhere('t.id != :id')
    ->setParameter('title', $tag->getTitle())
    ->setParameter('id', $tag->getId())
    ->getQuery()
;
$tags = $query->getResult();
```

QueryBuilder

Codebeispiel 123 *QueryBuilder*

Wir ermitteln mit dieser Abfrage alle Tags, die den gleichen Title, jedoch unterschiedliche IDs haben. Die zweite Bedingung ist übrigens zwingend nötig, da wir ja **andere** Tags finden wollen. In einem Controller würden wir so 11 Zeilen belegen, und was an dieser Stelle passiert, wäre nicht unbedingt sofort erkennbar. Eine Kürzung bzw. bessere Lesbarkeit wäre in der Datei, wo wir die Abfrage benötigen, möglich, auch wenn wir danach ingesamt mehr Code haben. Dieser Umstand ist jedoch bei OOP eigentlich der Normalfall. Zur Verbesserung der Lesbarkeit bietet uns Doctrine die Möglichkeit, eine eigene ***Repository-Klasse*** für eine Entity festzulegen und den Code unserer Datenbankabfrage dorthin auszulagern.

Beispiel

```php
<?php

namespace Repositories;

use Doctrine\ORM\EntityRepository;

class TagRepository extends EntityRepository
{
    public function findDuplicates(\Entities\Tag $tag)
    {
        $em = $this->getEntityManager();

        $query = $em
            ->createQueryBuilder()
            ->select('t')
            ->from('Entities\Tag', 't')
            ->where('t.title = :title')
            ->andWhere('t.id != :id')
            ->setParameter('title', $tag->getTitle())
            ->setParameter('id', $tag->getId())
            ->getQuery()
        ;

        return $query->getResult();
    }
}
```

src/Repositories/TagRepository.php

Codebeispiel 124 *src/Repositories/TagRepository.php*

In Zeile 3 nutzen wir den Namespace `Repositories`, der für alle unsere ***Repository***-Klassen gelten wird. Damit wir weiterhin Zugriff auf die aus Lektion 7 bekannten ***Repository***-Methoden haben, benutzen wir eine Vererbung von `Doctrine\ORM\EntityRepository`. Unsere Datenbankabfrage landet in der neuen Methode `TagRepository#findDuplicates()`. Da es sich um eine Methode handelt, ist sie nun wiederverwendbar und kann überall aufgerufen werden, wo wir Zugriff auf das ***Repository*** haben. Bei dem Parameter benutzen wir Type-Hinting, um uns die Benutzung dieser Methode beim Programmieren zu erleichtern. Wir erhalten also sofort eine Rückmeldung, wenn wir einen falschen Parameter benutzen. Achten Sie bitte darauf, den Namespace und einen führenden Backslash anzugeben, da wir uns derzeit im Namespace `Repositories` befinden.

Wie in Zeile 11 dargestellt, können wir in den Repository-Klassen eine Instanz des EntityManagers `$em` über die Methode

`EntityRepository#getEntityManager()` erhalten. Sie brauchen sich also bei den Controller-, Repository- und Validator-Klassen nur an eine einzige identische Vorgehensweise gewöhnen, da die Methode immer den gleichen Namen trägt.

Beispiel

```php
<?php

namespace Entities;

use Doctrine\ORM\Mapping as ORM;
use Webmasters\Doctrine\ORM\Util;

/**
 * @ORM\Entity(repositoryClass="Repositories\TagRepository")
 * @ORM\Table(name="tags")
 */
class Tag
{
    // gekuerztes Beispiel
}
```

src/Entities/Tag.php

Codebeispiel 125 *src/Entities/Tag.php*

Um die neue ***Repository***-Klasse benutzen zu können, müssen Sie deren Namen (inklusive Namespace) der Entity `Tag` über den Parameter `repositoryClass` in der Annotation `@ORM\Entity` mitteilen. Diese Annotation haben wir bereits verwendet, jedoch bisher ohne Parameter.

Die neue Methode `TagRepository#findDuplicates()` können wir jedoch nicht nur in unseren Controller-Klassen verwenden. Da wir beispielsweise in unseren neuen Validator-Klassen auch Zugriff auf die Repositories haben, ist hier ein Einsatz ebenfalls problemlos möglich.

Beispiel

```php
<?php

namespace Validators;

use Webmasters\Doctrine\ORM\EntityValidator;
```

```
 6
 7 class TagValidator extends EntityValidator
 8 {
 9     public function validateTitle($title)
10     {
11         $tag = $this->getEntity();
12
13         if (empty($title)) {
14             $this->addError('Das Feld Title ist leer.');
15         } elseif (strlen($title) < 3) {
16             $this->addError('Der Title sollte mindestens 3 Ze
17         } elseif ($this->getRepository()->findDuplicates($tag
18             $this->addError('Der Title existiert bereits.');
19         }
20     }
21 }
```

src/Validators/TagValidator.php (Version 2)

Codebeispiel 126 *src/Validators/TagValidator.php (Version 2)*

In Zeile 11 ermitteln wir zunächst das komplette zu validierende Objekt, da wir dieses als Parameter für unsere neue Methode benötigen. In Zeile 17 prüfen wir nun, ob wir Tags mit identischem Title, aber anderer ID finden können. Wir erhalten entweder ein leeres Array oder ein Array mit `Tag`-Objekten. Ein befülltes Array wird immer als `true` ausgewertet und somit wird nur in diesem Fall die Fehlermeldung ausgegeben.

13.3 Zufallsdatensätze

Kommen wir nun zu einem weiteren Einsatzzweck für eine eigene Repository-Methode. Doch überlegen wir zunächst, wie man bei MySQL nur mittels SQL-Code eine beliebige Anzahl von zufälligen Datensätzen (z. B. 5) ermittelt. Kommt Ihnen der nachfolgende Code bekannt vor?

Beispiel

```
SELECT * FROM tbl_name ORDER BY RAND() LIMIT 5;
```

ORDER BY RAND()

Codebeispiel 127 *ORDER BY RAND()*

Ich beziehe mich einmal kurz auf die offizielle Anleitung, in der sinngemäß steht, dass **ORDER BY RAND() in Kombination mit LIMIT nützlich zur Auswahl eines Zufallswertes aus einer Datensatzmenge ist**. Doch leider unterstützt Doctrine 2 bewusst keine Sortierung anhand dieser MySQL-Funktion. Diese Vorgehensweise ist nämlich bei tausenden von Datensätzen extrem unperformant. Liest man sich jedoch in der offiziellen MySQL-Anleitung die Kommentare zum `SELECT`-Statement durch, so findet man im Kommentar von »Dennis Lindkvist on June 28 2006« nachfolgende etwas performantere Variante.

Beispiel

```
SELECT *, RAND() AS rnd FROM tbl_name ORDER BY rnd LIMIT 5;
```

ORDER BY RAND() - Alternative

Codebeispiel 128 *ORDER BY RAND() - Alternative*

Dummerweise unterstützt Doctrine 2 die Funktion `RAND()` jedoch nirgendwo im DQL. Wir können diese aber mit einer benutzerdefinierten ***DQL-Funktion*** nachrüsten. Eine solche DQL-Funktion wird als Klasse nachgerüstet, die in diesem Fall bereits als `Webmasters\Doctrine\ORM\Query\RandFunction` in den Webmasters Doctrine Extensions enthalten ist. Der Aufruf mittels QueryBuilder könnte also theoretisch wie folgt lauten:

Beispiel

```
$query = $em
    ->createQueryBuilder()
    ->select('a, RAND() AS rnd')
    ->from('Entities\Article', 'a')
    ->orderBy('rnd')
    ->setMaxResults(5)
    ->getQuery()
;
$result = $query->getResult();
```

RAND() mit dem QueryBuilder

Codebeispiel 129 *RAND() mit dem QueryBuilder*

Praktisch quittiert Doctrine diesen Aufruf jedoch mit der Meldung `Error:`

`Expected known function, got 'RAND'`, wir müssen also noch irgendwie mitteilen, wo sich die betreffende Klasse befindet. Dies ist über verschiedene Wege möglich. Da der `QueryBuilder`-Aufruf wegen seiner Länge und schlechten Lesbarkeit sowieso in das `ArticleRepository` ausgelagert werden sollte, habe ich mich für die nachfolgend in den Zeilen 12-16 dargestellte Variante entschieden.

Beispiel

```php
<?php

namespace Repositories;

use Doctrine\ORM\EntityRepository;

class ArticleRepository extends EntityRepository
{
    public function findRandoms($max = 5)
    {
        $em = $this->getEntityManager();
        $config = $em->getConfiguration();
        $config->addCustomNumericFunction(
            'RAND',
            '\Webmasters\Doctrine\ORM\Query\RandFunction'
        );

        $query = $em
            ->createQueryBuilder()
            ->select('a, RAND() AS rnd')
            ->from('Entities\Article', 'a')
            ->orderBy('rnd')
            ->setMaxResults($max)
            ->getQuery()
        ;
        $result = $query->getResult();

        //Noetig wegen der speziellen Schreibweise von RAND(
        $entities = array();
        foreach ($result as $key => $val) {
            $entities[$key] = $val[0];
        }

        return $entities;
    }
}
```

src/Repositories/ArticleRepository.php

Codebeispiel 130 *src/Repositories/ArticleRepository.php*

In diesen Zeilen erweitern wir die Konfiguration unseres EntityManagers um die benötigte MySQL-Funktion bzw. Klasse. Als weitere Besonderheit finden Sie in den Zeilen 29-32 Code, der dem Umstand geschuldet ist, dass `$result` durch die spezielle `SELECT`-Schreibweise ein zweidimensionales Array enthält. In diesem Array sind die Objekte auf der zweiten Ebene mit dem Array-Schlüssel `0` hinterlegt. Außerdem finden Sie auf dieser zweiten Ebene unter dem Schlüssel `rnd` den benutzten Zufallswert.

Die Methode `BenutzerRepository#findRandoms()` hat einen Parameter `$max`, über den man die Anzahl der gewünschten zufälligen Datensätze bestimmen kann. Als Rückgabewert erhalten wir immer ein Array. Dieses kann entweder leer sein oder enthält maximal die gewünschte Anzahl von Datensätzen als Objekte, sofern hierfür genug Datensätze vorhanden sind.

13.4 Testen Sie Ihr Wissen

1. In welcher Datei kann klassenbasierter Validierungscode für unsere `User`-Entity abgelegt werden und wo wird diese Datei gespeichert?
2. Wozu werden eigene Repository-Klassen benutzt?

13.5 Aufgaben zur Selbstkontrolle

Übung 40:

Implementieren Sie das vorgestellte `TagRepository` und erweitern Sie den `TagValidator` um den vorgestellten dritten Fehlerfall. Doppelte Werte für `$title` sollten nun nicht mehr möglich sein. Testen Sie dies.

Übung 41:

Ergänzen Sie den noch fehlenden Code zur automatischen Datumsbefüllung in der Methode `Article#setPublishAt()`. Erstellen Sie einen neuen Datensatz und füllen Sie hierbei das optionale Formularfeld nicht aus. Wenn Sie diesen Datensatz danach editieren, so sollte das Feld einen Zeitpunkt als Wert enthalten.

Übung 42:

Nun müssen Sie sich noch um den `ArticleValidator` kümmern und ihn erweitern. Keines der mit Sternchen gekennzeichneten Pflichtfelder darf leer sein. Dies betrifft auch das Tagging, bei welchem Sie die Methode `ArrayCollection#isEmpty()` zur Validierung der Sammlung benötigen.

13.6 Optionale Aufgaben

Übung 43:

Die Standard-Aktion und die Aktion `search` des Controllers sortieren die Articles nach dem Erstellungsdatum. Leider gilt dies (noch) nicht für die Aktion `read` im `TagController`. Auch hier soll zukünftig der neueste Article zuerst angezeigt werden. Dies ist mit einer weiteren Annotation für sortierte Beziehungen möglich.

Machen Sie sich mit dieser Möglichkeit vertraut und setzen Sie sie im Newsticker um. Beachten Sie hierbei, dass wir einen Namespace bzw. Alias für unsere Annotationen und bei den Attributsnamen den lowerCamelCase verwenden.

Übung 44:

Implementieren Sie das vorgestellte `ArticleRepository` und testen Sie die Methode `ArticleRepository#findRandoms()`.

Hinweis: Ich habe mich bei der Musterlösung beispielsweise für die Anzeige eines zufälligen Article-Title auf der Startseite unter dem Suchformular entschieden. Diesen habe ich dann mit der Aktion `read` zur Anzeige der Detailseite verlinkt.

14 Eine Einführung in das Thema Sicherheit

In dieser Lektion lernen Sie
• wieso man beim Erstellen einer webbasierten Anwendung paranoid werden kann.

14.1 Absolute Sicherheit ist unmöglich

Im Bereich der Informationstechnologie gibt es den Spruch, dass Daten erst dann sicher sind, wenn sie gar nicht erst erhoben werden. Webbasierte Anwendungen kommen heutzutage jedoch nur noch selten ohne Benutzereingaben aus und immer wenn diese möglich sind, ist dies auch mit Sicherheitsproblemen verbunden.

Würden sich die erhobenen Daten auf einem normalen PC befinden, so könnten wir ihn ausschalten, sämtliche Kabel entfernen, ihn in einen Tresor packen, dessen Kombination mit verbundenen Augen ändern, ihn an einer wenig besuchten Stelle verstecken und uns dann hypnotisieren lassen, um die Position zu vergessen. Dies ist jedoch praktisch unmöglich und zudem könnte trotzdem ein versierter Safeknacker zufällig über ihn stolpern. Dies ist übrigens eine sehr freie Interpretation eines Zitats von Eugene Spafford, deren Aussage jedoch verständlich sein sollte: Eine absolute Sicherheit (einer webbasierten Anwendung) ist unmöglich. Alle unsere Bemühungen können stets nur als bestmöglicher Versuch gelten, da es unzählige Angriffsmöglichkeiten und Fallstricke gibt. Wir könnten den Rest unseres Lebens in die Entwicklung der ultimativ abgesicherten Anwendung investieren, nur damit uns ein auf einem kleinen gelben Zettel notiertes Kennwort zum Verhängnis wird. Wussten Sie übrigens, dass laut einer Umfrage im Jahr 2004 **71 Prozent** von 172 befragten Personen ihr firmeninternes Passwort für eine Tafel Schokolade preisgeben würden?

IT-Sicherheitsexperten sprechen deshalb immer nur von einem **Risiko-Management** und nie von einer Eliminierung. Wie viel Zeit und Aufwand man in dieses Management steckt, hängt auch vom Kontext ab. Eine Firma, die mit Kreditkartendaten arbeitet, sollte mehr investieren als eine bloggende Privatperson. Dies bedeutet natürlich nicht, dass ein Blog kein lohnendes Ziel für einen Angreifer ist.

> Marc Ruef, August 2005 in Marcs Blog: »Kann in jedes System eingebrochen werden? Ja. Ist jedoch ein System nach 48 Stunden nicht gefallen, explodiert der Aufwand und wird (...) unwirtschaftlich.« Sicherheit ist somit immer eine Kosten-Nutzen-Rechnung, sowohl auf Seite der Verteidiger als auch für die Angreifer.

Bedenken Sie aber auch, dass Angreifer unterschiedliche Motive haben können. Die Kosten-Nutzen-Rechnung eines Angreifers mit politischen oder persönlichen Motiven (z. B. ein verärgerter ehemaliger Angestellter) wird ganz anders aussehen als die eines finanziell motivierten.

Die aktuelle Lektion soll Ihnen einen Überblick über das Themengebiet »PHP-Sicherheit« bieten, kann jedoch aufgrund der vielfältigen Gefahren nicht mehr sein als eine Einführung und somit ein Ausgangspunkt für Ihre eigenen Recherchen.

14.2 Fünf empfehlenswerte Tugenden

Wenn man sich ein typisches Programmierprojekt ansieht, so gibt es einige Punkte, die ein Programmierer gerne weitestmöglich nach hinten verschiebt. Hierzu zählen ***Usability*** (dt. Bedienbarkeit), ***Refactoring*** (Codeüberarbeitung zur Ermöglichung einer besseren Wartbarkeit), Sicherheit und Dokumentation. Und wir wissen ja alle, dass am Projektende Zeit rar ist und deswegen auf manches verzichtet wird. Wie oft fehlt es beispielsweise an einer brauchbaren Dokumentation? Dies bedeutet übrigens nicht, dass der Programmierer schlecht ist. Er hat sich lediglich eine ungünstige Vorgehensweise angewöhnt. Auch kann es sein, dass diese Vorgehensweise gar nicht seine eigene Idee

war, sondern ihm vom Management vorgeschrieben wurde. Für viele Firmen geht es nämlich oftmals lediglich darum, schnellstmöglich einen Prototypen zu erstellen oder neue Features zu integrieren. Der verbleibende »unnütze« Rest, der nur Aufwand bedeutet und kein Geld einbringt, gerät dabei schnell ins Hintertreffen. Obwohl gerade dieser Rest langfristig sehr wahrscheinlich sogar Geld sparen würde.

Doch unsere Anwendungen laufen schon lange nicht mehr in einer streng begrenzten Umgebung mit einem genau bekannten Benutzerkreis, sondern sind jedem über das Internet verfügbar. Die meisten Benutzer unserer Web-Anwendungen haben lautere Absichten, doch es gibt immer ein paar Individuen, die unser bestes Gut wollen, und das sind unsere Daten bzw. die Daten unserer Kunden. Aus diesem Grund sollten sich sicherheitsbewusste Entwickler fünf Tugenden zu eigen machen. Die folgende Liste basiert auf »Pro PHP Security« von Snyder, Myer, Southwell S. 10ff. (siehe Abschnitt 15.3).

1. **Nichts ist zu 100 Prozent sicher.** Selbst wenn wir unsere Anwendung gegen alle bekannten Angriffsvarianten absichern, wird jemand eine neue entwickeln. Sicherheit kann also niemals als ein Feature einer Anwendung angesehen und in einem einzigen Arbeitsschritt abgehandelt werden, sondern wird immer wieder unsere Aufmerksamkeit und gegebenenfalls auch unsere Reaktion erfordern.
2. **Traue niemals deinen Benutzern und ihren Eingaben.** Ist das Ziel nur lohnend genug, so wird sich immer jemand mit böswilligen Absichten oder einfach jemand mit genug Neugierde und Zeit finden.
3. **Benutze immer mehrere Verteidigungslinien.** Je mehr Hürden ein Eindringling überwinden und je mehr Zeit er somit investieren muss, desto eher sucht er sich einfachere Gegner.
4. **Je wartbarer ein Code ist, desto einfacher kann man ihn absichern.** Wenn Sie sich ein Codefragment ansehen und es innerhalb einer Minute verstehen, dann können Sie es besser absichern, als wenn Sie erst etliche Stunden oder gar Tage zum Verständnis dieses Codes benötigen.
5. **Vier Augen sehen mehr als zwei.** Besprechen Sie Ihren Code in regelmäßigen Abständen mit einem anderen Programmierer. Ein solches Peer Reviewing fördert die Codequalität und verhindert weitestgehend eine

Verdrängung des »unnützen« Rests. Alternativ ist auch ein sogenanntes Pair Programming empfehlenswert, wobei zwei Programmierer am gleichen Rechner sitzen. Der eine Programmierer schreibt in diesem Fall den Code und der andere überprüft die neuen Codezeilen im Moment des Schreibens.

14.3 Security through Obscurity

Security through Obscurity ist ein äußerst umstrittenes Sicherheitskonzept, bei dem versucht wird, Sicherheit durch Unklarheit zu erreichen. Wenn diese Unklarheit (beispielsweise über den internen Aufbau einer Anwendung oder die verwendeten Versionen von Dritthersteller-Software) die einzige Verteidigungslinie einer Anwendung bildet, so ist die Strategie bereits zum Scheitern verurteilt. Es stimmt zwar, dass ein Angreifer umso mehr Zeit für die Suche nach Angriffsmöglichkeiten benötigt, je weniger Informationen er hat. Es muss jedoch auch klar sein, dass die Angriffsmöglichkeiten trotzdem vorhanden sind und nicht einfach verschwinden.

Das Konzept selbst ist übrigens ein Verstoß gegen ***Kerckhoffs Maxime***, welche auf die Programmierung übertragen aussagt, dass eine Anwendung aufgrund ihres Designs sicher sein sollte und nicht, weil das Design dem Angreifer unbekannt ist. Ein Zitat des Begründers der Informationstheorie *Claude Elwood Shannon* »The enemy knows the system«[16] (dt. »Der Feind kennt das System«) ist die Prämisse, unter der heutzutage Anwendungen entwickelt werden sollten. Es kann jedoch trotzdem nicht schaden, mit manchen Informationen (wie beispielsweise Versionsnummern) zu geizen.

---

16 Claude Elwood Shannon: Communication Theory of Secrecy Systems. In: Bell System Technical Journal. 28, Nr. 4, 1949, S. 656–715

Eine Web-Anwendung benötigt zum Betrieb neben dem eigenen Anwendungs-Code eine Vielzahl von Software-Komponenten:

- Betriebssystem (z. B. Linux oder Windows)
- Webserver-Software (z. B. Apache, nginx oder IIS)

- Datenbank Management System (z. B. MySQL)
- Skript-Interpreter/Programmiersprache (z. B. PHP, Python oder Ruby)
- Bibliotheken und Frameworks von Drittherstellern (z. B. Doctrine, jQuery, Symfony, Zend Framework, Django oder Ruby on Rails)
- eventuell wurde ja auch ein Content-Management-System (kurz CMS) als Basis eingesetzt (z. B. Drupal, TYPO3 oder WordPress).

Eine Informationsgewinnung über jede einzelne dieser Komponenten kann es einem Angreifer ermöglichen, eine potenzielle Schwachstelle zu identifizieren. Problematisch ist in diesem Zusammenhang, dass die meisten von ihnen in ihrer Standard-Konfiguration äußerst kommunikativ sind (Fehlermeldungen/Antworten bzw. spezielle Reaktionsweisen).

Nutzen wir für ein paar Beispiele den Browser *Firefox* und installieren die Plug-ins Wappalyzer und Live HTTP Headers.

> **Achtung**
>
> Beachten Sie, dass Wappalyzer in der Standard-Konfiguration Ihre Ergebnisse in anonymisierter Form an den Hersteller übermittelt. Sollten Sie dies nicht wünschen, so müssen Sie die Übermittlung in den Add-on-Einstellungen deaktivieren.

14.3.1 www.webmasters-fernakademie.de

Betrachten wir als erstes Beispiel die Startseite der *Webmasters Fernakademie*. Bei meinem letzten Test spuckte *Wappalyzer* folgende Infos aus:

- Betriebsystem: Ubuntu (eine kostenlose Linux-Distribution)
- Webserver-Software: Apache 2.4.7
- Programmiersprache: Ruby
- Frameworks: jQuery und Ruby on Rails

Wie gelangt das Add-on eigentlich an diese Daten? Öffnen Sie doch einmal in *Firefox* im Menü unter Extras das Add-on *Live HTTP Headers* und rufen Sie die Startseite erneut auf. Scrollen Sie dann in der Anzeige der HTTP-Header nach ganz oben zur ersten Server-Antwort (engl. Response).

Beispiel

```
HTTP/1.1 200 OK
Date: Wed, 30 Sep 2015 14:39:56 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: Phusion Passenger 5.0.15
(...)
```

Server-Response

Codebeispiel 131 *Server-Response*

Der Webserver selbst ist also bereits bei dieser Antwort ziemlich kommunikativ und teilt sowohl das benutzte Betriebssystem als auch die Webserver-Software im sogenannten ***Server-Banner*** mit. Weitere Informationen wie das verwendete JavaScript-Framework kann man beispielsweise in Erfahrung bringen, indem man den JavaScript-Code einfach nach dem Wort *jQuery* durchsucht.

14.3.2 blog-das-oertchen.de

Betrachten wir als zweites Beispiel mein privates Blog und dieses Mal als erstes die Antwort des Servers beim Aufruf der Startseite.

Beispiel

```
HTTP/1.1 200 OK
Date: Wed, 30 Sep 2015 14:42:52 GMT
Server: Apache
(...)
```

Server-Response

Codebeispiel 132 *Server-Response*

Der Webserver ist weit weniger auskunftsfreudig und gibt in seiner Antwort

lediglich preis, dass Apache eingesetzt wird.

- Webserver-Software: Apache
- Programmiersprache: PHP
- Frameworks: jQuery
- CMS: Drupal

Mir ist zwar nicht bekannt, wie *Wappalyzer* intern arbeitet, aber das CMS und das JavaScript-Framework sind in diesem Fall unter anderem am HTML-Quelltext (Inhalt des `head`-Tags) erkennbar. Da das CMS Drupal in PHP geschrieben wurde, lässt dies eine einfache Schlussfolgerung auf die verwendete Programmiersprache zu.

14.3.3 www.google.de

Bei der Startseite von *Google* fördert *Wappalyzer* keinerlei brauchbare Informationen zutage. *Live HTTP Headers* ist in diesem Fall leider auch nur bedingt hilfreich.

Beispiel

```
HTTP/2.0 200 OK
Date: Wed, 30 Sep 2015 14:44:43 GMT
Server: gws
(...)
```

Server-Response

Codebeispiel 133 *Server-Response*

Wie man dieser Antwort entnehmen kann, nutzt Google als Webserver-Software nicht *Apache*, sondern den selbstentwickelten *gws* bzw. Google Web Server. Über diese angeblich auf *Linux* basierende Software sind kaum Informationen bekannt. Google nutzt hier also anscheinend die Unklarheit als (zusätzliches) Sicherheits-Konzept.

14.3.4 localhost

Wenden wir uns abschließend unserer lokalen Entwicklungsplattform zu. In der Standard-Konfiguration sind *XAMPP* und Konsorten äußerst kommunikativ. Nach der Aktualisierung meiner XAMPP-Installation auf PHP 7 erhielt ich beispielsweise nachfolgende Daten:

- Betriebsystem: Windows
- Webserver-Software: Apache 2.4.23
- Programmiersprache: PHP 7.0.9

Beispiel

```
HTTP/1.1 200 OK
Date: Tue, 25 Oct 2016 12:43:21 GMT
Server: Apache/2.4.23 (Win32) OpenSSL/1.0.2h PHP/7.0.9
(...)
```

Server-Response

Codebeispiel 134 *Server-Response*

Alle diese Informationen stammen übrigens aus der Antwort des Servers. Wenn man diese genauer betrachtet, erfährt man sogar, dass ich eine 32-Bit Version von Apache benutze.

14.3.5 Server-Konfiguration

Diese Unterlagen behandeln eigentlich keine Themen aus dem Bereich der Administration. Trotzdem möchte ich Ihnen eine kurze Übersicht über die mir bekannten Konfigurationsparameter anbieten. Dies kann jedoch nicht mehr sein als ein erster Ansatzpunkt zur Beendigung dieser Offenherzigkeit.

Parameter	Wert	Datei
ServerTokens	Full | OS | Minimal | Minor | Major | Prod	httpd.conf
ServerSignature	On | EMail | Off	httpd.conf
expose_php	On|Off	php.ini

Tabelle 14.1 *Konfigurationsparameter*

Der erste Wert ist jeweils die Standard-Einstellung und der letzte Wert meine persönliche Empfehlung. Wenden Sie sich am besten an Ihren Administrator, um die entsprechenden Anpassungen vornehmen zu lassen. Wenn Sie normalen Webspace und keinen dedizierten Server benutzen, so ist eine Anpassung sehr wahrscheinlich nicht möglich. Trotz all dieser Einstellungen ist übrigens die Angabe `Apache` immer noch im Server-Banner enthalten. Es ist zwar möglich, die verwendete Webserver-Software mit weiteren Maßnahmen zu tarnen, doch gibt es sogenannte Webserver Fingerprinting Tools, die anhand spezifischer Verhaltensweisen (beispielsweise der Reihenfolge der Header) trotzdem erfolgreich eine Schlussfolgerung auf das verwendete Produkt ziehen können. Dies ist auch nicht weiter schlimm, da heutzutage jedes sogenannte Skriptkiddie in der Lage ist, mit gebrauchsfertigen Tools ganze Listen von Websites in einem Rutsch anzugreifen. Diese Tools versuchen dann automatisiert eine Vielzahl von Angriffen und decken dabei die unterschiedlichsten Komponenten und Lücken ab. Stellen Sie sich am besten als Analogie einen Schützen mit einer Schrotflinte vor, der ohne zu zielen auf das WWW schießt.

Gegen diese ungezielten Angriffe kann man wenig machen, außer die eigene Software stets aktuell zu halten und bekanntgewordene Lücken sofort zu schließen. Allerdings gibt es auch eine Vielzahl von zielgerichteten Angriffen, und hier kann eine durch uns erreichte Unklarheit den Angreifer Zeit kosten. Je länger er sich jedoch mit uns beschäftigt, desto wahrscheinlicher wird seine Entdeckung.

Es kann also sinnvoll sein, einem solchen Angreifer die Informationsgewinnung zu erschweren. Hängen Sie beispielsweise einmal den Dateinamen *composer.json* oder *composer.lock* an die URL Ihres Projektverzeichnisses an, so werden Sie feststellen, dass diese Dateien problemlos im Browser anzeigbar sind. Dies liegt daran, dass der PHP-Interpreter lediglich Dateien mit bestimmten Endungen[17] parst und Dateien mit unbekannter Endung vom Webserver sofort als `text/plain` ausgeliefert werden.

---

17 Standard ist die Endung *.php*, in der *httpd.conf* können jedoch weitere Endungen erlaubt werden.

Sofern eine PHP-Datei lediglich Funktions- bzw. Datenlieferant für eine andere Datei ist und manuell per `require(_once)` oder `include(_once)` in

diese eingebunden wird, sollten Sie (sofern möglich) zur besseren Unterscheidbarkeit eine gesonderte Dateiendung für die eingebundene Datei verwenden. Empfehlenswert ist hierbei beispielsweise eine doppelte Dateiendung. Diese sollte jedoch möglichst auf *.php* enden. Zwei Beispiele haben Sie mit *.inc.php* und *.tpl.php* bereits kennengelernt.[18]

18 Backup-Dateien (z. B. *.bak* oder *.old*) sollten auf einem Produktivserver niemals zu finden sein, auch nicht mit doppelter Dateiendung.

Die Composer-Dateien haben jedoch festgelegte Namen, weswegen eine Umbenennung nicht in Frage kommt. Wir müssen also einen anderen Ansatz wählen, um beispielsweise die Inhalte der *composer.json* vor neugierigen Blicken zu schützen. Sonst könnte jeder Angreifer sehr simpel relativ detaillierte Informationen über den Aufbau unseres Projektes erlangen.

Beispiel

```
# Browser-Zugriff verbieten
<Files composer.*>
    # Apache <= 2.2
    <IfModule !mod_authz_core.c>
        Order deny,allow
        Deny from all
    </IfModule>
    # Apache >= 2.3
    <IfModule mod_authz_core.c>
        Require all denied
    </IfModule>
</Files>
```

.htaccess (Version 1 - Blacklist)

Codebeispiel 135 *.htaccess (Version 1 - Blacklist)*

Bei festgelegten Dateinamen ist es ratsam, dem Browser einfach den kompletten Zugriff auf diese Dateien zu verbieten. Sofern Sie einen Apache-Webserver nutzen, ist dies meistens über eine sogenannte *.htaccess*-Datei möglich. Auf ganz billigem Webspace fehlt diese Möglichkeit leider oftmals.

Die dargestellte *.htaccess*-Datei verbietet den Browser-Zugriff auf alle Dateien mit dem Namen *composer* und einer beliebigen Dateiendung. Da die verbotenen Dateien festgelegt werden, handelt es sich hierbei um den sogenannten

Blacklist-Ansatz (dt. schwarze Liste). Eine solche Blacklist erfordert jedoch eine sehr genaue Kenntnis aller problematischen Dateien; deswegen ist der entgegengesetzte ***Whitelist***-Ansatz (dt. weiße Liste) in der Regel empfehlenswerter.

Wie man an der Beispiel-Datei erkennen kann, gibt es zwei unterschiedliche Schreibweisen: Eine Schreibweise bis Apache 2.2 und eine Schreibweise ab Apache 2.3 (allerdings wurde Apache 2.3 nie als stabile Version veröffentlicht). Spätestens seit dem Release von Apache 2.4.1 am 17. Februar 2012 wird jedoch die bisherige Syntax nur noch unterstützt, sofern das Apache-Modul *mod_access_compat* aktiv ist. Dies ist zwar bei aktuellen XAMPP-Versionen standardmäßig der Fall, aber nicht zwingend bei allen Hostern bzw. Apache-Servern. Gewöhnen Sie sich also besser direkt daran, dass es zwei Schreibweisen gibt. Die 2.2er Syntax ohne aktives *mod_access_compat* führt übrigens auf einem Apache 2.4 zu einem 500er Serverfehler. Um dies zu vermeiden, sollte man prüfen, ob die neue auf *mod_authz_core* basierende Syntax unterstützt wird oder nicht.

Beispiel

```
# Browser-Zugriff komplett verbieten
# Apache <= 2.2
<IfModule !mod_authz_core.c>
    Order deny,allow
    Deny from all
</IfModule>
# Apache >= 2.3
<IfModule mod_authz_core.c>
    Require all denied
</IfModule>

# Browser-Zugriff selektiv erlauben
<FilesMatch "((^$)|(^(index\.php|.+\.(js|css|gif|jpe?g|png))$))">
    # Apache <= 2.2
    <IfModule !mod_authz_core.c>
        Allow from all
    </IfModule>
    # Apache >= 2.3
    <IfModule mod_authz_core.c>
        Require all granted
    </IfModule>
</FilesMatch>
```

.htaccess (Version 2 - Whitelist)

***Codebeispiel 136** .htaccess (Version 2 - Whitelist)*

Ein solcher Whitelist-Ansatz ist wesentlich restriktiver und somit sicherer. Die dargestellte Whitelist erlaubt beispielsweise lediglich Zugriff auf Dateien mit dem Namen *index.php* und Dateien mit den aufgeführten Dateiendungen (js, css, gif, jpg, jpeg oder png). Oftmals kennt man zwar zu Projektstart nicht alle benötigten Endungen, man kann diese aber problemlos nach und nach ergänzen. Schalten Sie aber bitte nicht kommentarlos jede Datei(-endung) auf Zuruf frei.

Die beiden bisher gezeigten Varianten zur Limitierung des Browser-Zugriffs funktionieren leider nicht bei jedem Webspace, da hierfür in der *.htaccess* die Direktiven zur Steuerung des Zugriffs von Hosts erlaubt sein müssen (`AllowOverride Limit`). Ist dies nicht der Fall, so scheitert der Browser-Zugriff mit einem 500er Serverfehler. Ich möchte Ihnen deswegen noch eine alternative Vorgehensweise zeigen.

Beispiel

```
<IfModule mod_rewrite.c>
   RewriteEngine On
   RewriteRule !(^|index\.php|\.(js|css|gif|jpe?g|png))$ - [L
</IfModule>
```

.htaccess (Version 3 - Whitelist)

***Codebeispiel 137** .htaccess (Version 3 - Whitelist)*

Diese Variante funktioniert allerdings nur, sofern das Apache-Modul *mod_rewrite* auf dem Webserver aktiv ist. Dies ist beispielsweise bei *XAMPP* standardmäßig nicht der Fall, man kann das Modul allerdings nachträglich aktivieren. Im Gegensatz zu den beiden vorherigen Varianten gibt es bei einem inaktiven Modul keinen 500er Serverfehler. Stattdessen sind beispielsweise die Composer-Dateien einfach weiterhin im Browser aufrufbar. Testen Sie bei dieser Variante also immer, ob die Limitierung des Browser-Zugriffs auch wirklich funktioniert.

Übung 45:

1. Schützen Sie die Composer-Dateien mittels der dritten Variante gegen

einen Browser-Zugriff.

2. Testen Sie, ob Sie noch Composer-Dateien wie beispielsweise *composer.json* oder *composer.lock* im Browser aufrufen können. Sollte dies noch der Fall sein, müssen Sie *mod_rewrite* aktivieren oder die zweite Variante verwenden!
3. Testen Sie, ob Sie im Browser noch PHP-Dateien mit einem anderen Namen als *index.php* (z. B. *reset.php*) aufrufen können. Dies sollte nicht (mehr) möglich sein. Um Datenbank-Änderungen mit der *setup.php* oder *reset.php* umzusetzen, müssen Sie also fortan die Datei *.htaccess* erstmal entfernen und danach wiederherstellen.

Eine Verschleierung von (veralteten) Versionsangaben entbindet den Betreiber und Entwickler einer Anwendung natürlich trotzdem nicht davon, regelmäßig Sicherheits-Updates der verwendeten Software-Komponenten vorzunehmen. Mit dem *Security Advisories Checker* von *SensioLabs* ist es beispielsweise möglich, die verwendeten Composer-Pakete auf bekannt gewordene Sicherheitslücken zu überprüfen.

14.4 Parametermanipulation

Schauen wir uns nun noch einmal das Template unserer Startseite genauer an.

Beispiel

```
<div class="links">
    [ <a
        href="index.php?action=read&id=<?php echo $article->
    >Details</a> ]
    [ <a
        href="index.php?action=edit&id=<?php echo $article->
    >Edit</a> ]
    [ <a
        href="index.php?action=delete&id=<?php echo $article
    >Delete</a> ]
</div>
```

templates/IndexController/indexAction.tpl.php (Ausschnitt)

Codebeispiel 138 *templates/IndexController/indexAction.tpl.php (Ausschnitt)*

Wenn Sie sich den Ausschnitt ansehen, so werden Sie feststellen, dass mehrfach eine ID als Link-Parameter ergänzt wird. Dies ist beispielsweise beim Details-Link der Fall. Doch was ist, wenn ein Besucher diese ID-Angabe manipuliert? Dies muss nicht einmal ein böswilliger Angreifer, sondern kann genauso gut ein neugieriger Besucher sein, der einfach mal die ID auf 99 erhöht. Sofern die ID nicht existiert, erhält er die Meldung `Fatal error: Call to a member function getTitle() on a non-object in (...)\templates\IndexController\readAction.tpl.php on line 3`. Diese Meldung offenbart gleich drei Details über unsere Anwendung: Wir nutzen objektorientierte Programmierung, unsere Templates liegen im Ordner *templates* und das Template hat einen namentlichen Zusammenhang zur aktuellen Aktion im URL-Parameter. Eine solche Schwachstelle bezeichnet man auch als unerwünschte Informationspreisgabe (engl. ***Unplanned Information Disclosure***).

14.4.1 Whitelist - Variante 1

Doch wie lässt sich dieses Problem lösen und gleichzeitig die Usability unserer Anwendung etwas verbessern? Wären die Articles unveränderliche Daten und kämen niemals neue Datensätze hinzu, so könnten wir einfach die erlaubten IDs als Whitelist in unserem Code hinterlegen.

Beispiel

```php
<?php

namespace Controllers;

class IndexController extends AbstractBase
{
    // gekuerztes Beispiel

    public function readAction()
    {
        $whitelist = array(1, 2, 3, 4);
        if (!in_array($_GET['id'], $whitelist)) {
            die('ID nicht vorhanden!');
        }
```

```
15
16          $em = $this->getEntityManager();
17          $article = $em
18              ->getRepository('Entities\Article')
19              ->find($_GET['id'])
20          ;
21
22          $this->addContext('article', $article);
23      }
24
25      // gekuerztes Beispiel
26  }
```

src/Controllers/IndexController.php (Version 1 - statische Whitelist)

Codebeispiel 139 *src/Controllers/IndexController.php (Version 1 - statische Whitelist)*

Unveränderliche datenbankbasierte Daten begegnen uns in einer Web-Applikation jedoch so gut wie nie. Doch selbst wenn dies der Fall wäre, so wäre die Wartbarkeit des Codes relativ schlecht, da man im Falle einer Anpassung die Datenbankinhalte und Zeile 11 im Code verändern müsste. Und eine solche Änderungsanforderung kommt meist schneller als man hofft.

14.4.2 Whitelist - Variante 2

Wir benötigen also eine andere Variante unserer Whitelist, bei der wir nicht die bekannten IDs im Code hinterlegen, sondern auf die vom DBMS gelieferten Daten reagieren.

Beispiel

```
 1  <?php
 2
 3  namespace Controllers;
 4
 5  use Doctrine\ORM\EntityManager;
 6
 7  abstract class AbstractBase
 8  {
 9      // gekuerztes Beispiel
10
11      public function render404()
12      {
13          header('HTTP/1.0 404 Not Found');
14          die('Error 404');
```

```
15      }
16
17      // gekuerztes Beispiel
18  }
```

src/Controllers/AbstractBase.php (Version 1 - Ausschnitt)

Codebeispiel 140 *src/Controllers/AbstractBase.php (Version 1 - Ausschnitt)*

```
 1  <?php
 2
 3  namespace Controllers;
 4
 5  class IndexController extends AbstractBase
 6  {
 7      // gekuerztes Beispiel
 8
 9      public function readAction()
10      {
11          $em = $this->getEntityManager();
12          $article = $em
13              ->getRepository('Entities\Article')
14              ->find($_GET['id'])
15          ;
16
17          $article || $this->render404();
18
19          $this->addContext('article', $article);
20      }
21
22      // gekuerztes Beispiel
23  }
```

src/Controllers/IndexController.php (Version 1b - flexible Whitelist)

Codebeispiel 141 *src/Controllers/IndexController.php (Version 1b - flexible Whitelist)*

Sie werden sich eventuell über die Schreibweise in Zeile 17 des Controllers wundern. Dies ist eine sehr verkürzte Schreibweise einer `if`-Abfrage, welche sich die als Lazy Evaluation bekannte Auswertungstechnik von PHP zunutze macht. Der zweite Teil dieser Oder-Bedingung wird nämlich nur ausgeführt, wenn der Finder anhand der ID keinen Datensatz findet und deswegen `null` als Rückgabewert zurückliefert.

> Planen Sie zu Beginn der Anwendungsentwicklung unbedingt auch Prüfroutinen für die verwendeten Parameter.

Haben Sie gedacht, dass der ID-Parameter nun sicher gegen sämtliche

Manipulationsversuche ist? Sie haben sich geirrt, denn wenn man den Parameter komplett entfernt und nur die Aktionsangabe übrig lässt, so erhält man eine Reihe von informativen PHP-Meldungen. Beispielsweise erfährt man durch diese, dass Doctrine mit der Klasse `Entities\Article` verwendet wird. Dieses Problem haben wir bereits in »Band 1: Grundlagen der OOP« beim Thema MVC für den Aktions-Parameter gelöst; wir haben dabei den Trinitäts-Operator verwendet. Ich persönlich bin hierzu ehrlich gesagt zu faul und kombiniere eine kürzere Methode mit der Deaktivierung der Fehleranzeige. Denken Sie also unbedingt daran, im **produktiven Umfeld** den `debug_mode` in der *default-config.php* zu deaktiveren.

Beispiel

```
 1 <?php
 2
 3 namespace Controllers;
 4
 5 class IndexController extends AbstractBase
 6 {
 7     // gekuerztes Beispiel
 8
 9     public function readAction()
10     {
11         $em = $this->getEntityManager();
12         $article = $em
13             ->getRepository('Entities\Article')
14             ->find((int)$_GET['id'])
15         ;
16
17         $article || $this->render404();
18
19         $this->addContext('article', $article);
20     }
21
22     // gekuerztes Beispiel
23 }
```

src/Controllers/IndexController.php (Version 1c)

Codebeispiel 142 *src/Controllers/IndexController.php (Version 1c)*

Was passiert nun? In Zeile 14 erzwingen wir den Datentyp Integer. Fehlt der URL-Parameter komplett, so wird hierdurch die ID `0` erzwungen. Diese gibt es aber bei keinem normalen Datensatz, und somit greift die Whitelist in Zeile 17. Wir erhalten natürlich trotzdem noch die Hinweis-Meldung für den **nicht**

vorhandenen Array-Index, aber nur sofern wir die Fehleranzeige (noch) nicht deaktiviert haben und überhaupt jemand die URL-Parameter manipuliert.

> Trauen Sie niemals den Eingaben und Angaben eines Benutzers. Hierzu zählen beispielsweise URL-Parameter und sämtliche Formulardaten (auch die von versteckten Feldern und Radio-Buttons). URL-Parameter können problemlos direkt in der URL-Angabe manipuliert werden, und bei der Manipulation von Formulardaten helfen Browser-Plugins wie beispielsweise Firebug.

14.4.3 Dateiangaben als Parameter

Betrachten wir nun noch einmal den grundsätzlichen Ablauf in unserer Anwendung:

1. Ein Benutzer ruft die Anwendung über den Front-Controller (mit oder ohne URL-Parameter) auf.
2. Bei einem korrekten Controller-Parameter in der URL wird eine Controller-Klasse instanziiert und die Methode `AbstractBase#run()` in diesem Objekt aufgerufen.
3. Der Aktions-Parameter landet dann im Attribut `$template` dieser Instanz, wobei er unter anderem mit dem Suffix `Action` und der Dateiendung *.tpl.php* ergänzt wird.
4. Sofern der Aktions-Parameter einer existierenden Methode in der Controller-Klasse zugeordnet werden kann, wird diese aufgerufen.
5. Durch die Methode `AbstractBase#render()` wird abschließend das Template *layout.tpl.php* per `require_once` eingebunden.
6. Dieses Layout-Template bindet wiederum per `require_once` das eigentliche Template der Aktion ein, wobei sich Letzteres in einem Controller-spezifischen Unterordner von *templates* befindet.

Bei dem ganzen Ablauf gehen wir davon aus, dass die Template-Anzeige nur in der vorgesehenen Art und Weise über die beiden URL-Parameter manipuliert

werden kannn, da wir ja den Pfad und die Dateiendung fest vorgeben. Vor PHP 5.3.4 war dies jedoch leider nicht der Fall, da weitergehende Manipulationen mit sogenannten ***Null Bytes*** möglich waren. Hierdurch hätte der PHP-Interpreter beispielsweise bei uns das Suffix `Action` und die Dateiendung *.tpl.php* ignoriert.

Beispiel

```php
<?php

namespace Controllers;

use Doctrine\ORM\EntityManager;

abstract class AbstractBase
{
    // gekuerztes Beispiel

    public function run($action)
    {
        $this->addContext('action', $action);

        $methodName = $action; // . 'Action';
        $this->setTemplate($methodName);

        if (method_exists($this, $methodName)) {
            $this->$methodName();
        } else {
            //$this->render404();
        }

        $this->render();
    }

    public function render404()
    {
        header('HTTP/1.0 404 Not Found');
        die('Error 404');
    }

    // gekuerztes Beispiel

    protected function setTemplate($template, $controller = 
    {
        if (empty($controller)) {
            $controller = $this->getControllerShortName();
        }

```

```
41          $this->template = $controller . '/' . $template; //
42      }
43
44      // gekuerztes Beispiel
45
46      protected function render()
47      {
48          extract($this->context);
49
50          $message = $this->getMessage(); // Get flash message
51          $template = $this->getTemplate();
52
53          require_once $this->basePath . '/templates/layout.tp
54      }
55  }
```

src/Controllers/AbstractBase.php (Version 2 - Ausschnitte)

Codebeispiel 143 *src/Controllers/AbstractBase.php (Version 2 - Ausschnitte)*

Gehen wir für einen kleinen Test davon aus, dass wir bei einer fehlenden Methode kommentarlos den restlichen Code der Controller-Klasse ausführen. Wir kommentieren Zeile 21 deshalb einfach aus. Manipulieren Sie testweise den Aktions-Parameter Ihrer Anwendung mit der Angabe `action=../../composer.json%00`. Wird dies **nicht** mit einem Fatal error quittiert, so ist Ihr PHP-Interpreter noch anfällig für eine ***Remote File Inclusion*** mittels ***Null Byte Termination***. Dies sollte heutzutage jedoch eigentlich nicht mehr der Fall sein.

Um trotzdem ein Resultat sehen zu können, emulieren wir das Problem mit den Null Bytes, indem wir das Suffix (Zeile 15) und die Dateiendung (Zeile 41) nicht mehr ergänzen. Mit der Angabe `action=../../composer.json` erhalten wir nun eine Ausgabe der *composer.json*.

Schlimmstenfalls kann über eine solche ***Remote File Inclusion*** sogar Code von einem fremden Webserver ausgeführt werden (Stichwort `allow_fopen_url`). Bedenken Sie bei der Absicherung von Include- und Require-Operationen aber auch, dass böswilliger PHP-Code durch Benutzer-Uploads auf Ihren Webserver gelangen und sich durchaus innerhalb einer Grafik-Datei befinden kann.

Da man sofern möglich mehrere Verteidigungslinien benutzen sollte, sollten Sie sich nicht nur auf die gefixte PHP-Version verlassen. Brechen Sie bei einer fehlenden Methode für die Aktion immer die Verarbeitung komplett ab. Diese

Umsetzung einer Whitelist nutzen Sie theoretisch seit »Band 1: Grundlagen der OOP«, wo Sie im Rahmen einer Aufgabe erarbeitet werden sollte. Zudem sollten Angaben wie Ordner, Suffixe und Dateiendungen fest in Ihrem Code verankert sein und nicht über URL-Parameter übergeben werden können.

Beispiel

```
 1 <?php
 2
 3 require_once 'inc/functions.inc.php';
 4 require_once 'inc/helper.inc.php';
 5
 6 require_once 'inc/bootstrap.inc.php';
 7
 8 // Session needed for flash messages
 9 session_start();
10
11 // Path to our index.php
12 $basePath = dirname(__FILE__);
13
14 $controller = isset($_GET['controller']) ? $_GET['controller
15 $controller = preg_replace('/[^a-z]/', '', $controller);
16
17 $action = isset($_GET['action']) ? $_GET['action'] : 'index'
18 $action = preg_replace('/[^a-z]/', '', $action);
19
20 $controllerNamespace = 'Controllers\\';
21 $controllerName = $controllerNamespace . ucfirst($controller
22
23 if (class_exists($controllerName)) {
24     $requestController = new $controllerName($basePath, $em)
25     $requestController->run($action);
26 } else {
27     $requestController = new Controllers\IndexController($ba
28     $requestController->render404();
29 }
```

index.php

Codebeispiel 144 *index.php*

Als zusätzliche Verteidungslinie erlauben wir mittels einer ***Filterung*** in Zeile 15 und 18 nur noch Kleinbuchstaben in den Variablen `$controller` und `$action`. Alle Zeichen, die dieser Whitelist nicht entsprechen (wie beispielsweise der von uns zum Verzeichniswechsel verwendete Slash `/`), werden einfach gelöscht. Somit sind die Variablen nun wirklich komplett abgesichert und `$action` kann

auch problemlos mit `echo` in unseren Templates ausgegeben werden. Bei `$controller` ist eine solche Template-Ausgabe derzeit übrigens nicht möglich, da wir die Variable nicht an unsere Templates weitergeben.

> Verwenden Sie in Prüfroutinen von Include- und Require-Operationen möglichst immer einen Whitelist-Ansatz, um eine ***Parametermanipulation*** zu verhindern.

Übung 46:

1. Schützen Sie den Newsticker mit der vorgestellten Error404-Whitelist gegen eine Manipulation von ID-Parametern. Dies ist überall dort sinnvoll, wo ein einzelner Datensatz anhand seiner ID ausgelesen werden soll und die ID-Angabe aus der URL oder aus Formulardaten stammt. Im Zusammenhang mit der Methode `Article#addTag()` ist dies jedoch meiner Meinung nach unnötig, da hier eine Manipulation bereits durch das Type-Hinting verhindert wird. Allerdings schadet es nichts, wenn Sie mittels `find((int)$id)` trotzdem einen Integer für die IDs erzwingen.
2. Integrieren Sie danach auch noch die zeichenbasierte Whitelist für `$controller` und `$action` im Front-Controller.

14.5 Die bekanntesten Angriffsvarianten

Sie wissen nun, dass die Parameter Ihrer Anwendung durch Manipulationen angreifbar sind, und wie Sie zum Schutz Prüfroutinen verwenden können. Dabei können Sie entweder auf eine Blacklist-Variante setzen oder auf eine Whitelist, was meist sinnvoller ist. Kommen wir nun zu den bekanntesten Angriffsvarianten.

14.5.1 SQL-Injections

Eine ***SQL-Injection*** (dt. SQL-Einschleusung) ist, wie der Name schon sagt, eine Angriffsmethode im Zusammenhang mit einem SQL-basierten Datenbank Management System. Bei einer SQL-Injection werden jedoch keine Sicherheitslücken des DBMS ausgenutzt, sondern es wird eine spezielle Variante der Parametermanipulation verwendet. Ziel ist es, die SQL- oder DQL-Anweisungen in unserem Code mit benutzerdefiniertem Code zu erweitern. Dies kann einem Angreifer jedoch nur gelingen, wenn unsere Anwendung dies zulässt.

Angriffsszenarien

In meiner Tätigkeit als Tutor begegne ich beispielsweise immer wieder Varianten des nachfolgenden Codes:

Beispiel

```php
<?php

namespace Controllers;

class IndexController extends AbstractBase
{
    // gekuerztes Beispiel

    public function searchAction()
    {
        $like = $_POST['like'];

        $em = $this->getEntityManager();
        $query = $em
            ->createQueryBuilder()
            ->select('a')
            ->from('Entities\Article', 'a')
            ->where("a.title LIKE '%$like%'")
            ->orderBy('a.createdAt', 'DESC')
            ->getQuery()
        ;

        //var_dump($query->getDQL());

        $articles = $query->getResult();

        $this->addContext('articles', $articles);
        $this->setTemplate('indexAction');
```

```
29         }
30     }
```

src/Controllers/IndexController.php (Version 2 - SQL-Injections)

Codebeispiel 145 *src/Controllers/IndexController.php (Version 2 - SQL-Injections)*

Der Code sieht eigentlich ganz harmlos aus und soll einen einfachne Suchfilter ermöglichen. Hierzu wird in Zeile 11 zunächst die Variable `$like` mit dem Suchbegriff befüllt. In Zeile 18 lässt die Umsetzung dann SQL-Injections zu, da sie diese Benutzereingabe direkt in die `WHERE`-Bedingungen des QueryBuilders und somit in den DQL-Code übernimmt.

Testen Sie die Lücke doch einmal mit dem Suchbegriff `honk' OR 1=1 OR a.title LIKE '`. Betrachten Sie anschließend die entstandene DQL-Abfrage, indem Sie den `var_dump()` in Zeile 23 aktivieren.

```
SELECT a FROM Entities\Article a WHERE a.title LIKE '%honk' OR
```

SQL-Injection

Codebeispiel 146 *SQL-Injection*

Es ist uns also gelungen, die WHERE-Bedingung der SELECT-Anweisung mit eigenem Code zu erweitern. Durch das `OR 1=1` haben wir zudem erreicht, dass alle Datensätze der Tabelle `articles` ausgelesen werden. Bei einer Suche scheint dies nicht so schlimm zu sein, doch was ist, wenn Sie beispielsweise einen ähnlichen Code bei Ihrer Login-Prüfung verwenden?

> In dem Beispiel wurde `$_POST` manipuliert und hierfür ein `text`-Eingabefeld benutzt. Dies bedeutet natürlich nicht, dass dies bei anderen Daten-Quellen (z. B. `$_GET` oder `$_SERVER`) und Feld-Typen (z. B. `hidden` oder `radio`) nicht auch möglich wäre.

SQL-Injections sollten bei der ORM-Bibliothek von Doctrine nur bei `SELECT`-Statements vorkommen können, da wir ja normalerweise für `INSERT`, `UPDATE` und `DELETE` keinen eigenen DQL-Code schreiben müssen.

Lösungsansätze

Kommen wir jetzt zur guten Nachricht: Solange Sie für **alle** Benutzereingaben ***Prepared Statements*** mit (benannten) Platzhaltern verwenden, müssen Sie sich um SQL-Injections keine Sorgen machen.

Beispiel

```
1  <?php
2
3  namespace Controllers;
4
5  class IndexController extends AbstractBase
6  {
7      // gekuerztes Beispiel
8
9      public function searchAction()
10     {
11         $like = $_POST['like'];
12
13         $em = $this->getEntityManager();
14         $query = $em
15             ->createQueryBuilder()
16             ->select('a')
17             ->from('Entities\Article', 'a')
18             ->where('a.title LIKE :search')
19             ->setParameter('search', '%' . $like . '%')
20             ->orderBy('a.createdAt', 'DESC')
21             ->getQuery()
22         ;
23
24         //var_dump($query->getDQL());
25
26         $articles = $query->getResult();
27
28         $this->addContext('articles', $articles);
29         $this->setTemplate('indexAction');
30     }
31 }
```

src/Controllers/IndexController.php (Version 2b - Prepared Statement)

Codebeispiel 147 *src/Controllers/IndexController.php (Version 2b - Prepared Statement)*

Wir müssen lediglich in Zeile 18 den Wert für `LIKE` durch einen (benannten) Platzhalter ersetzen. In Zeile 19 befüllen wir dann diesen Platzhalter und verketten hierbei die beiden `%`-Wildcards mit der Benutzereingabe. Wir dürfen die Wildcards übrigens nicht im DQL-Statement notieren, da sonst die

automatisch gesetzten Anführungszeichen zu Problemen führen würden.

> ***Prepared Statements*** schützen Ihre Anwendung gegen SQL-Injections und wirklich nur dagegen!

Leider kann man Platzhalter nur für Werte und nicht für Spaltennamen einsetzen. Möchte man also beispielsweise eine Benutzereingabe in einem `ORDER BY` verwenden, so sollte man zur Absicherung eine Whitelist (beispielsweise mit einem im Code hinterlegten Array) nutzen.

Übung 47:

1. Integrieren Sie den aktuellen Stand der Suche.
2. Überprüfen Sie diese Implementierung. Benutzen Sie hierbei auch testweise einen nicht vorhandenen Suchbegriff wie beispielsweise `_x_`.
3. Überprüfen Sie den Code Ihres Newstickers auf weitere potenzielle SQL-Injection-Lücken und fixen Sie diese. Eigentlich dürfte jedoch maximal die Suche anfällig gewesen sein.

14.5.2 Cross-Site-Scripting

Anders als bei SQL-Injections ist bei einem Angriff mittels ***Cross-Site-Scripting*** (kurz XSS) nicht unsere Anwendung auf dem Webserver das Ziel, sondern der Schadcode richtet sich gegen unsere Besucher bzw. deren Browser. XSS-Lücken sind vorhanden, wenn eine Anwendung Daten von einem Benutzer annimmt und diese danach unverändert im Browser (eines anderen Benutzers) anzeigt. So ist es nämlich einem Angreifer problemlos möglich, seinen in einer clientseitigen Skriptsprache (z. B. JavaScript oder Visual Basic Script) erstellten Code in den HTML-Quelltext unserer Anwendung einzuschleusen. Gelingt eine XSS-Attacke, so sind nahezu beliebige DOM-Manipulationen möglich (z. B. Änderungen an Texten und Links), womit beispielsweise Aktionen wie Phishing oder ein dauerhaftes Defacement eingeleitet werden können. Im schlimmsten Fall ermöglicht eine solche Lücke einem Angreifer sogar einen sogenannten Drive-by-Download (die Installation

von Software auf dem Rechner des Besuchers). Anfang 2013 war beispielsweise die Website von *NBC* Mitverursacher eines solchen Angriffs. Eine XSS-Attacke ist meist breit gestreut und nicht zielgerichtet auf eine bestimmte Person gerichtet, obwohl dies theoretisch auch möglich wäre. Das perfide an einer solchen Attacke ist, dass das Opfer unserer Website vertraut und dieses Vertrauen vom Angreifer für seine Zwecke missbraucht wird.

Angriffsszenarien

Heutige Websites bestehen in den seltensten Fällen nur aus von uns erstellten Texten bzw. Inhalten. Meistens gibt es zusätzlich sogenannte nutzergenerierte Inhalte (z. B. Kommentare oder Kunden-Meinungen), die vielfach sofort nach der Speicherung für die restlichen Besucher zur Verfügung stehen. Eine Manipulation dieser Inhalte ist dann auch der einfachste Angriffspunkt für ein sogenanntes **persistentes** Cross-Site-Scripting.

Ein anderer Angriffsvektor ist das **reflektierte** (engl. reflected) Cross-Site-Scripting. Es nutzt beispielsweise eine in einem Suchformular enthaltene XSS-Lücke aus, die sich auf die Anzeige der Suchergebnisse auswirkt. Diese Lücke wäre, wenn man lediglich die eigene Website betrachtet, relativ ungefährlich, da nur der Angreifer selbst den manipulierten HTML-Quelltext ausgeliefert bekäme. Allerdings ist es bei vielen Ergebnisseiten möglich, diese zu bookmarken. Hierfür werden alle Sucheingaben in `$_GET` übermittelt und befinden sich somit in der URL. Dank der populären sozialen Netzwerke geht ein ansprechend präsentierter oder mit einem Kurz-URL-Dienst wie Bit.ly getarnter Link jedoch schnell um die Welt und erreicht viele potenzielle Opfer. In Kombination mit einem Wurm kann sich solch ein schädlicher Link sogar noch schneller verbreiten. Ein gutes Beispiel ist ein XSS-Wurm, der sich 2006 auf StudiVZ verbreitete und dem Diebstahl der Logindaten diente.

Beispiel

```
<?php if (($action == 'index') || ($action == 'search')): ?>
    <form id="search" action="index.php" method="get">

        <input type="hidden" name="action" value="search" />

        <label for="like">Title like</label>
```

```
        <input name="like" id="like" type="text" />

        <input type="submit" class="button" value="Abschicke

    </form>
<?php endif; ?>

<!-- gekuerztes Beispiel -->

<?php if (($action == 'search') && !empty($like)): ?>
    <p>
        Sie suchten nach:
        &raquo;<?php echo $like; ?>&laquo;
    </p>
<?php endif; ?>

<?php foreach ($articles as $article): ?>
    <!-- gekuerztes Beispiel -->
<?php endforeach; ?>
```

templates/IndexController/indexAction.tpl.php (Version 1 - Ausschnitt)

Codebeispiel 148 *templates/IndexController/indexAction.tpl.php (Version 1 - Ausschnitt)*

```
<?php

namespace Controllers;

class IndexController extends AbstractBase
{
    // gekuerztes Beispiel

    public function searchAction()
    {
        $like = $_GET['like'];

        $em = $this->getEntityManager();
        $query = $em
            ->createQueryBuilder()
            ->select('a')
            ->from('Entities\Article', 'a')
            ->where('a.title LIKE :search')
            ->setParameter('search', '%' . $like . '%')
            ->orderBy('a.createdAt', 'DESC')
            ->getQuery()
        ;

        $articles = $query->getResult();

        $this->addContext('articles', $articles);
```

```
27          $this->addContext('like', $like);
28          $this->setTemplate('indexAction');
29      }
30  }
```

src/Controllers/IndexController.php (Version 3)

Codebeispiel 149 *src/Controllers/IndexController.php (Version 3)*

Um die grundsätzliche Vorgehensweise zu demonstrieren, habe ich unsere Suche mit einer XSS-Lücke versehen. Diese Lücke besteht aus zwei Komponenten:

1. Der Controller akzeptiert in Zeile 11 der *IndexController.php* den Suchbegriff nun aus der Superglobalen `$_GET` und nicht mehr aus `$_POST`. Dies ist übrigens nicht grundsätzlich schlecht und kann sogar gewünscht sein, da man nur so im Browser einen Bookmark für die Suchergebnisse erstellen kann. Um die Superglobale `$_GET` zu nutzen, wurde auch das Attribut `method` im `form`-Tag unseres HTML-Codes (Zeile 2) angepasst und die Übergabe des `action`-Parameters in ein verstecktes Formularfeld verschoben (Zeile 4).
2. Der Suchbegriff wird (unverändert) in Zeile 19 des Templates *indexAction.tpl.php* ausgegeben.

Testen Sie die anfällige Anwendung mit den nachfolgenden Angaben:

1. Suchbegriff: `<script type="text/javascript">alert("XSS")</script>`
2. *index.php?action=search&like=<script>alert("XSS")<%2Fscript>*

Ein Angreifer wird natürlich keine Alertbox anzeigen wollen, sondern beispielsweise eine Phishing-Aktion einleiten, indem er Sie auf einen anderen Server umleitet.

Beispiel

```
1 window.location = 'http://www.google.com/';
2 document.getElementById('search').action = 'http://www.google
```

JavaScript-Umleitungen

Codebeispiel 150 *JavaScript-Umleitungen*

Mit dem JavaScript-Code in Zeile 1 würde die aktuell angezeigte Webseite

sofort mit der Startseite von Google ersetzt. Dies wäre lediglich ein Ärgernis für unseren Besucher. Doch was ist, wenn ein Angreifer unsere Website optisch exakt nachbaut und unseren Besucher auf dieser Kopie zum Login verleitet? Dann hätte unsere Anwendung eine erfolgreiche Phishing-Attacke ermöglicht und wir vermutlich das Vertrauen des Besuchers in unsere Anwendung verspielt. Genauso verhält es sich mit dem JavaScript-Code in Zeile 2, mit dem kleinen Unterschied, dass der Besucher erst bei der Benutzung unseres Suchformulars umgeleitet würde.

Lösungsansätze

Wenn die Eingaben eines Benutzers für eine Anzeige im Browser benötigt werden, so sollte diese Anzeige nur nach einer Ausfilterung oder Maskierung problematischer Angaben erfolgen. Bedenken Sie herbei jedoch, dass sich JavaScript nicht nur in einem `script`-Tag verbergen kann.

Beispiel

```
1 <script type="text/javascript">alert("XSS")</script>
2 <a href="#" onmouseover="alert('XSS')">Klick mich</a>
3 <iframe src=javascript:alert('XSS')></iframe>
```

JavaScript-Einbindung

Codebeispiel 151 *JavaScript-Einbindung*

Dies sind nur ein paar Beispiele, die bei meinen Tests mit Firefox (Version 20) problemlos funktionierten, die selbst jedoch lediglich die Spitze des Eisbergs bilden. Doch wie können wir solche Angriffe verhindern? Im Falle unserer Suche ist dies relativ simpel, da beim Suchbegriff keines der drei HTML-Tags überhaupt unterstützt werden muss. Um genau zu sein, braucht das Eingabefeld keinerlei HTML-Tags zu erlauben. Wir können diese also bei der Ausgabe mit der Funktion `strip_tags()` komplett herausfiltern.

Beispiel

```
1 <?php
2
3 function e($dirty)
4 {
```

```
5       echo strip_tags($dirty);
6   }
```

inc/helper.inc.php (Version 1)

Codebeispiel 152 *inc/helper.inc.php (Version 1)*

```
1  <?php if (($action == 'index') || ($action == 'search')): ?>
2      <form id="search" action="index.php" method="get">
3
4          <input type="hidden" name="action" value="search" />
5
6          <label for="like">Title like</label>
7          <input
8              name="like" id="like" type="text"
9              <?php if (!empty($like)): ?>
10                 value="<?php e($like); ?>"
11             <?php endif; ?>
12         />
13
14         <input type="submit" class="button" value="Abschicke
15
16     </form>
17 <?php endif; ?>
18
19 <!-- gekuerztes Beispiel -->
20
21 <?php if (($action == 'search') && !empty($like)): ?>
22     <p>
23         Sie suchten nach:
24         &raquo;<?php e($like); ?>&laquo;
25     </p>
26 <?php endif; ?>
27
28 <?php foreach ($articles as $article): ?>
29     <!-- gekuerztes Beispiel -->
30 <?php endforeach; ?>
```

templates/IndexController/indexAction.tpl.php (Version 2 - Ausschnitt)

Codebeispiel 153 *templates/IndexController/indexAction.tpl.php (Version 2 - Ausschnitt)*

Die Funktion `e()` soll Ihnen einiges an Schreibarbeit ersparen und gehört in den Bereich der sogenannten Views-***Helper***. Anstatt bei jeder Ausgabe von Benutzereingaben `echo strip_tags($wert)` schreiben zu müssen, reicht ein einfaches `e($wert)` aus. Wir weichen hier übrigens ausnahmsweise von unseren Programmierrichtlinien aus »Band 1: Grundlagen der OOP« ab, da mir einfach kein anderer sinnvoller kurzer Bezeichner eingefallen ist. Der Code wird nicht nur sicherer durch die ***Filterung***, sondern auch schlanker. Dank des

Einsatzes unseres neuen Helpers kann mit keinem der drei obigen Beispiele mehr eine Alertbox angezeigt werden.

Um die Usability unserer Suche zu verbessern, habe ich jedoch in Zeile 10 des Templates *indexAction.tpl.php* zusätzlich eine Ausgabe des Suchbegriffs im `value`-Attribut ergänzt. Wenn Sie nun den Suchbegriff mit integriertem JavaScript erneut ausprobieren, offenbart sich hierdurch ein weiteres Problem. Das erste doppelte Anführungszeichen des Strings `XSS` beendet nämlich die Ausgabe im `value`-Attribut.

Testen Sie doch einmal Ihre Anwendung mit den nachfolgenden Angaben:

1. Suchbegriff: `" style="border: 1px solid red`
2. Suchbegriff: `">Defacement`

Unser Code erlaubt also trotz ***Filterung*** im Helper immer noch einzelne (schließende) spitze Klammern und Anführungszeichen, was für ein ***Website-Defacement*** ausgenutzt werden kann. Um diese Zeichen an einem unerwünschten »Herauswuchern« zu hindern, ergänzen wir im Helper `e()` einen Aufruf der Funktion htmlspecialchars() und somit ein sogenanntes ***Escaping*** (dt. Maskierung) der Ausgabe.

Beispiel

```
1  <?php
2
3  function e($dirty, $encoding = 'UTF-8')
4  {
5      echo htmlspecialchars(
6          strip_tags($dirty),
7          ENT_QUOTES | ENT_HTML5,
8          $encoding
9      );
10 }
```

inc/helper.inc.php (Version 1b)

Codebeispiel 154 *inc/helper.inc.php (Version 1b)*

Da auch einfache Anführungszeichen an manchen Stellen zu einem Problem werden könnten, benutzen wir den optionalen zweiten Parameter `$flags` von

`htmlspecialchars()` und übergeben hier den Wert `ENT_QUOTES`, um auch diese Anführungszeichen in HTML-Entitäten umzuwandeln. Durch `ENT_HTML5` und `'UTF-8'` legen wir außerdem den von uns verwendeten HTML-Standard und Zeichensatz fest.

Leider ist das Leben nicht immer so einfach wie beim Beispiel unserer Suche. Oftmals muss beispielsweise HTML-Code im Rahmen einer Beschreibung erlaubt und ein XSS-Angriff trotzdem verhindert werden. Um dies zu lösen, sollten Sie **keinesfalls** auf die Idee kommen, Ausnahmen über den optionalen zweiten Parameter von `strip_tags()` zu ermöglichen. Jedes der erlaubten Tags unterstützt nämlich weiterhin die Nutzung von JavaScript-Event-Handlern. Wir benötigen stattdessen einen sogenannten ***XSS-Cleaner*** zur Bereinigung der Benutzereingaben. Eine XSS-Blacklist ist hierbei meiner Meinung nach wegen der unzähligen Ansätze und Varianten ein zum Scheitern verurteilter Ansatz und ich kann nur jedem Entwickler die Verwendung einer XSS-Whitelist (auch ***Purifier*** genannt) ans Herz legen.

Beispiel

```
{
    "autoload": {
        "psr-0": {
            "Controllers": "./src/",
            "Entities": "./src/",
            "Repositories": "./src/",
            "Validators": "./src/"
        }
    },
    "require": {
        "php": ">=5.4",
        "doctrine/orm": "2.5.5",
        "gedmo/doctrine-extensions": "2.4.24",
        "webmasters/doctrine-extensions": "4.0.0",
        "ezyang/htmlpurifier": "4.8.0"
    }
}
```

composer.json

Codebeispiel 155 *composer.json*

Soweit ich weiß, ist die Library *HTML Purifier* die bekannteste und erprobteste XSS-Whitelist. Allein über Packagist wurde sie bereits über 500.000 Mal

installiert.

Beispiel

```php
<?php

function e($dirty, $encoding = 'UTF-8')
{
    echo htmlspecialchars(
        strip_tags($dirty),
        ENT_QUOTES | ENT_HTML5,
        $encoding
    );
}

function purify($dirty)
{
    $config = HTMLPurifier_Config::createDefault();
    $purifier = new HTMLPurifier($config);
    echo $purifier->purify($dirty);
}
```

inc/helper.inc.php (Version 2)

Codebeispiel 156 *inc/helper.inc.php (Version 2)*

Verwenden Sie den Helper `purify()` immer dann, wenn eine Benutzereingabe ausgegeben werden soll, die HTML-Code unterstützen muss. Beachten Sie jedoch, dass der Einsatz von `e()` wesentlich performanter ist. Diese Funktion ist meiner Meinung nach sogar so performant, dass ich `echo` einfach komplett mit ihr ersetzt habe. Wenn Ihnen die Standard-Whitelist für Tags und Attribute nicht zusagt, so können Sie diese mit etwas Aufwand im Helper anpassen. Unter https://github.com/kennberg/php-htmlpurfier-html5 finden Sie übrigens ein entsprechendes Beispiel zur Aktivierung einiger HTML5-Tags. Ich persönlich hatte jedoch noch keinen Fall, wo ich diese in einer Benutzereingabe erlauben musste.

> *HTML Purifier* macht meines Wissens die Ausgabe von Benutzereingaben mit HTML-Code XSS-sicher. 100%ig garantieren wird dies jedoch wohl niemand. Deswegen sollten Sie die Library stets auf dem aktuellsten Stand halten, falls doch einmal ein Schlupfloch gefunden wird.

Übung 48:

1. Installieren Sie die Library *HTML Purifier* in Ihrem *vendor*-Verzeichnis.
2. Integrieren Sie die beiden neuen Helper in der *helper.inc.php* in Ihrem *inc*-Verzeichnis.
3. Integrieren Sie den aktuellen Stand der Suche. Vergessen Sie hierbei nicht, das Template *indexAction.tpl.php* anzupassen.
4. Schützen Sie auch den restlichen Newsticker, indem Sie für die Ausgabe von Werten aus unseren Entities und allen sonstigen Benutzereingaben einen der beiden neuen Views-Helper anstatt `echo` benutzen.

   Hinweis: Der Helper `e()` kann nicht in Kombination mit `nl2br()` eingesetzt werden, da sonst die gerade erzeugten `br`-Tags sofort wieder entfernt würden.

14.5.3 Weitere Angriffsmöglichkeiten

Bevor wir zu einem ganz anderen Thema übergehen, möchte ich noch kurz auf die Variante der CSRF-Angriffe (***Cross-Site Request Forgery***) eingehen. Hierbei handelt es sich im weitesten Sinn um das Gegenteil einer XSS-Attacke. Anstatt mittels Schadcode den Browser eines Besuchers anzugreifen, werden die im Browser hinterlegten Informationen für eine Veränderung von gespeicherten Daten auf unserem Webserver missbraucht. Bei einer lohnenswerten Manipulation kann es sich im schlimmsten Fall um eine Überweisung oder einen Kaufvorgang handeln, aber auch eine Löschung von Datensätzen in unserer Anwendungs-Datenbank kann durchaus von Interesse sein. Der Angreifer missbraucht also das Vertrauen einer Anwendung in einen »bekannten« Benutzer.

Sicherheitsrelevante Aktionen sollten grundsätzlich nur möglich sein, wenn der Benutzer bereits eingeloggt ist. Wird dieser Umstand bei einem CSRF-Angriff ausgenutzt, so handelt es sich um sogenanntes ***Session Riding***.

Eine CSRF-Lücke existiert immer dann, wenn eine Veränderung von Daten komplett über `$_GET` ausgelöst werden kann. Dies ist beispielsweise der Fall, wenn wir die Eingaben in einem Formular zwar mit der Methode `post` verschicken, im Controller aber `$_REQUEST` für die Verarbeitung verwenden. Wir sollten also bei der Verwendung von `$_REQUEST` grundsätzlich sehr vorsichtig sein.

Übung 49:

Überlegen Sie, welche Methode in unserer Controller-Klasse `IndexController` derzeit für einen CSRF-Angriff anfällig ist.

Und? Haben Sie herausgefunden, welche Methode anfällig ist? Nein? Dann schauen Sie noch einmal genau nach, wo derzeit überall die Methode `EntityManager#flush()` aufgerufen wird. An einer Stelle ist dieser Methoden-Aufruf **nicht** an ein befülltes `$_POST`-Array gekoppelt.

Beispiel

```
1 <img src="index.php?action=delete&id=1" style="display: none;'
2 <div style="background-image: url(index.php?action=delete&id=1
```

CSRF-Beispiele

Codebeispiel 157 *CSRF-Beispiele*

Doch wie lässt sich dieses Problem lösen?

Beispiel

```
 1 <p>
 2     Wollen Sie den Article
 3     &amp;quot;<?php e($article->getTitle()); ?>&amp;quot;
 4     vom <?php e($article->getCreatedAt()->format('d.m.Y')); ?
 5     wirklich entfernen?
 6 </p>
 7
 8 <form action="index.php?action=delete" method="post">
 9
10     <input name="id" type="hidden" value="<?php e($article->g
11
12     <input type="submit" class="button" value="Ja" />
```

```
13      <a href="index.php">Abbrechen</a>
14
15  </form>
```

templates/IndexController/deleteAction.tpl.php

Codebeispiel 158 *templates/IndexController/deleteAction.tpl.php*

```
 1  <?php
 2
 3  namespace Controllers;
 4
 5  class IndexController extends AbstractBase
 6  {
 7      // gekuerztes Beispiel
 8
 9      public function deleteAction()
10      {
11          $em = $this->getEntityManager();
12          $article = $em
13              ->getRepository('Entities\Article')
14              ->find((int)$_REQUEST['id'])
15          ;
16
17          $article || $this->render404();
18
19          if ($_POST) {
20              $em->remove($article);
21              $em->flush();
22
23              $this->setMessage('Article wurde entfernt.');
24              $this->redirect();
25          }
26
27          $this->addContext('article', $article);
28      }
29  }
```

src/Controllers/IndexController.php (Version 4)

Codebeispiel 159 *src/Controllers/IndexController.php (Version 4)*

Wir bauen eine Sicherheitsabfrage auf PHP-Basis. Hierzu ermitteln wir zunächst einmal die ID aus `$_REQUEST` anstatt `$_GET`. Danach verhindern wir die Ausführung des restlichen Codes unserer Aktion mit einer `if`-Abfrage. Das eigentliche Löschen wird also nur noch ausgeführt, sofern Formulardaten in `$_POST` vorliegen. Als Template benutzen wir ein einfaches Formular, das lediglich die ID des Datensatzes als verstecktes Formularfeld übermittelt. Wir haben also zwei Quellen für die ID und verwenden deswegen `$_REQUEST`.

Darüber geben wir den eigentlichen Text der Sicherheitsabfrage aus und bieten aus Gründen einer besseren Usability neben dem Submit-Button des Formulars noch einen Link zum Abbrechen des Vorgangs an.

> Erlauben Sie Änderungen in Ihrer Datenbank nur, wenn diese durch Daten in `$_POST` ausgelöst wurden. Dies können Sie beispielsweise mittels einer in PHP umgesetzten Sicherheitsabfrage lösen. Eine JavaScript-Abfrage (z. B. mit `confirm()` ist hierfür nicht ausreichend, kann jedoch durchaus als doppelte Sicherheitsabfrage vor einem Löschen nützlich sein.

Übung 50:

Schützen Sie den Newsticker mit der vorgestellten Sicherheitsabfrage auf PHP-Basis.

Diese Maßnahme ist jedoch bei Weitem noch kein 100%iger Schutz gegen CSRF-Angriffe. Ein Angreifer könnte beispielsweise über eine XSS-Lücke einen `iframe` mit einem Formular zu unserer Website hinzufügen und dieses dann per JavaScript und der Methode `post` verschicken. Für einen wirksameren Schutz müssen Sie stets sicherstellen, dass ein Request auch wirklich vom betreffenden Benutzer ausgelöst wurde. Deswegen sollten Sie zusätzlich folgende Ansätze bei der Planung Ihrer Anwendung berücksichtigen:

- Formular-Tokens
- erneute Kennworteingabe bei sicherheitskritischen Aktionen, wie es beispielsweise Amazon beim Start eines Bestellvorgangs (engl. Checkout) macht.
- Einmalkennwörter oder Transaktionsnummern (TANs), die im Bereich des Online-Bankings sehr verbreitet sind.
- eine Bestätigung per E-Mail mit einem Link zum Auslösen der Aktion. Eine solche E-Mail wird häufig bei vergessenen Passwörtern und einem damit verbundenen Kennwortreset genutzt. In dem Link sollte immer ein Einmalkennwort enthalten sein. Ohne dieses Kennwort darf das Auslösen der Aktion nicht möglich sein und durch diese Auslösung muss das

Kennwort »verbraucht« werden, damit die Aktion nicht erneut ausgelöst werden kann.

- CAPTCHAs
- Aufteilung von komplexen Vorgängen in einzelne Schritte (z. B. Buchungstunnel).

Zum Abschluss dieses Themenbereichs möchte ich Ihnen empfehlen, sich auch mit den weniger bekannten Angriffsmöglichkeiten vertraut zu machen. Bedenken Sie auch, dass ein Angriff aus einer Kombination mehrerer Techniken bestehen kann. Zum Einstieg bieten sich folgende Quellen an:

1. http://php.net/de/security
2. https://www.sitepoint.com/8-practices-to-secure-your-web-app/
3. https://www.sitepoint.com/top-10-php-security-vulnerabilities/
4. https://www.owasp.org/index.php/Category:Attack.

14.6 Authentisierungssicherheit

Zusätzlich zu möglicherweise in Ihrem Code verborgenen Sicherheitslücken gibt es noch weitere Fallstricke, die Sie bei der Umsetzung einer mit einem Login geschützten Anwendung bedenken sollten.

14.6.1 Passwörter

Der Bereich der Kennwörter scheint nicht wirklich in diese Lektion zu passen, da es sich zunächst einmal um kein technisches Problem, sondern um ein menschliches auf Seiten unserer Benutzer handelt. Doch wenn Sie an den Abschnitt 14.1 zurückdenken, werden Sie schnell merken, dass aus diesem menschlichen Problem schnell eine Gefahr für unsere Anwendung erwachsen kann. Wir sollten also technische Maßnahmen ergreifen, um diese Gefahr zu minimieren.

Vermeidung von unsicheren Passwörten

Unsichere Kennwörter sind ein Themenbereich, mit dem sich jeder Entwickler vertraut machen sollte. Denn hat ein Angreifer erst einmal gültige Logindaten ermittelt, so kann er den kompromittierten Account voll ausnutzen. Im einfachsten Fall wird er lediglich ein neues Passwort vergeben und den eigentlichen Benutzer so aussperren. So gewinnt er Zeit, kann sich die zugänglichen Daten in Ruhe zu Gemüte führen und diese, sofern er genügend Rechte erlangt hat, beliebig manipulieren. Hat der Angreifer zunächst einen Account mit sehr eingeschränkten Rechten erwischt, so kann er trotzdem versuchen, diesen für weitergehende Angriffe (z. B. mittels Cross-Site-Scripting) zu verwenden.

> Trauen Sie keinen Benutzereingaben, auch nicht denen eines eingeloggten Benutzers.

Um die Gefahr eines solchen Angriffs zu minimieren, sollten wir als Entwickler die Verwendung von unsicheren Passwörtern verhindern. Hierbei gibt es zwei Ansätze.

1. Unsere Anwendung vergibt serverseitig ein sicheres Kennwort (beispielsweise im Rahmen der Registrierung).
2. Die Anwendung unterstützt unsere Benutzer bei der Wahl eines sicheren Kennworts. Sicher ist hierbei natürlich relativ zu sehen, da eine 100%ige Sicherheit nicht erreichbar ist.

Beide Varianten sind in der freien Wildbahn ungefähr gleich häufig anzutreffen und haben jeweils einen klaren Vorteil. Die serverseitige Variante erzeugt (sofern korrekt umgesetzt) die wesentlich sichereren (komplett zufälligen) Passwörter, und die andere Variante hat einen höheren Benutzerkomfort, da man sich selbst festgelegte Kennwörter meist besser merken kann. Die serverseitige Variante missfällt mir persönlich, da man seine Benutzer durch vorgegebene Kennwörter schnell verärgern kann. Durch eine Kombination beider Varianten könnte man dies jedoch theoretisch etwas abmildern. Wir werden uns im Folgenden jedoch nur mit der zweiten Variante beschäftigen, da wir damit auch das Thema der Validierungen vertiefen können.

Bevor wir uns mit der genauen Implementierung beschäftigen, benötigen wir

zunächst ein Regelset für sichere Kennwörter. Würden wir unsere Benutzer nämlich komplett in Eigenregie walten lassen, so würden wir als Ergebnis Passwörter wie beispielsweise **Passwort** oder **geheim** erhalten. Diese Beispiele wären nicht einmal einen Fetzen Papier wert, um sie darauf zu notieren, da sie innerhalb von Sekunden durch einen sogenannten Wörterbuchangriff ermittelbar sind. Der Name dieser Angriffsweise basiert ursprünglich darauf, dass viele Menschen für ein neues Kennwort Begriffe aus einem Wörterbuch (Duden oder Lexikon, engl. dictionary) wählen. Das Oxford English Dictionary enthält beispielsweise ca. 600.000 Einträge. Heutzutage sind solche »Wörterbücher« jedoch sehr viel ausgereifter, da im Rahmen diverser Angriffe immer wieder tatsächlich verwendete Passwörter ergänzt werden konnten. Eine der größten Ergänzungen ermöglichte beispielsweise 2009 eine erfolgreiche SQL-Injection beim Online-Spiele-Dienst RockYou.com, durch welche eine Liste mit 32 Millionen Klartext-Kennwörtern ihren Weg in das Internet fand (nach Dubletten-Entfernung verblieben 14,3 Millionen). Je mehr dieser häufigen Passwörter wir also durch ein Regelset verhindern, desto länger benötigt ein Angreifer für einen erfolgreichen Treffer. Bei der Erstellung eines solchen Regelsets müssen wir allerdings immer einen brauchbaren Kompromiss zwischen der Sicherheit der Passwörter und dem Komfort der Benutzer finden.

Je länger ein Kennwort ist, desto höher fällt die benötigte Zeit für einen Angriff mit der sogenannten ***Brute-Force-Methode*** aus. Bei der Brute-Force-Methode werden im Gegensatz zu einem Wörterbuchangriff einfach alle möglichen Zeichenkombinationen ermittelt und ausprobiert, was mit einem deutlich höheren Zeitaufwand verbunden ist. Aus dieser Erkenntnis ergibt sich direkt unsere erste Regel:

1. **Mindestlänge**: Empfehlenswert ist laut dem Bundesamt für Sicherheit in der Informationstechnik eine Mindestlänge von **12 Zeichen** (Stand Oktober 2015). Geben Sie niemals eine genaue Länge vor, sondern immer nur eine Mindestanzahl oder einen Bereich. So haben die Benutzer mehr Spielraum bei der Auswahl. Bedenken Sie aber auch, dass die Merkbarkeit bei vielen Benutzern mit steigender Länge sinkt und eine weitere Erhöhung somit kontraproduktiv sein kann.
2. **Zulässige Zeichen**: Aus je mehr Zeichen ein Kennwort besteht, desto

größer fällt die Anzahl der möglichen Zeichenkombinationen aus. Würde man beispielsweise lediglich die Ziffern 0 bis 9 zulassen und gäbe eine vierstellige Pin vor, so gäbe es 10 * 10 * 10 * 10 = 10.000 Kombinationen. Würden wir hingegen die ursprüngliche 7-Bit ASCII-Tabelle erlauben, so stünden uns schon 95 (druckbare) Zeichen zur Auswahl und es ergäben sich 95 * 95 * 95 * 95 = 81.450.625 Kombinationen.

3. **Zeichendiversifikation**: Passwörter wie beispielsweise **1111**, **1234** oder **abcd** sind leider weit verbreitet und wenig sinnvoll, da sie einfach zu erraten sind. Ein sicheres Kennwort sollte deshalb Groß- und Kleinbuchstaben, Zahlen und Sonderzeichen und zu mindestens 50 Prozent unterschiedliche Zeichen enthalten.
4. **Persönliche Daten**: Durch die sozialen Netzwerke ist eine große Anzahl von Informationen über uns frei zugänglich, doch auch über andere Quellen sind viele Daten leicht herauszufinden (sogenanntes Social Engineering). Eine sinnvolle Implementierung sollte also auf jeden Fall verhindern, dass die der Anwendung bekannten Profildaten im Kennwort verwendet werden können. Es wäre zwar auch sinnvoll, Kennwörter auf Basis unserer Partner, Kinder oder Haustiere zu unterbinden, doch diese Informationen liegen unseren Anwendungen im Normalfall nicht vor.

Die obige Liste basiert auf »Sichere Webanwendungen mit PHP« von Wassermann S. 108f. (siehe Abschnitt 15.3).

Ein sicheres Kennwort

Machen wir uns nun einmal kurz Gedanken, wie wir nach diesen Regeln selbst ein sicheres Kennwort für uns erstellen würden. Dabei sollten wir zunächst bedenken, dass die RockYou-Liste viel über die Denkweise eines Benutzers bei der Erstellung eines Kennworts zeigte. So kamen nahezu alle Großbuchstaben am Anfang und Interpunktionszeichen wie das Komma oder der Punkt am Ende. Zudem gab es einen starken Trend zu Vornamen, gefolgt von einer Jahreszahl (z. B. **Andrea1984**). All dies sollten wir also vermeiden. Beginnen wir unseren Gedankengang mit einem einfachen Satz: »Ein PHP-Entwickler mag jQuery, aber er liebt Doctrine.« Daraus könnten wir zunächst das Passwort `EP-Emj,aelD.` entwickeln, indem wir lediglich jeden ersten Buchstaben und die Sonderzeichen berücksichtigen. In dieser ersten Variante beginnen wir

allerdings immer noch mit einem Großbuchstaben, und es fehlen noch Zahlen.

Um beide Probleme zu lösen, möchte ich das Thema Leetspeak in den Raum stellen. Hierbei werden einzelne Zeichen durch eine ähnlich aussehende Entsprechung ersetzt. Da eine komplette Ersetzung die Anzahl der erlaubten Zeichen unnötig reduzieren würde, würde ich lediglich eine Ersetzung aller ungeraden Vokalbuchstaben (ohne die deutschen Umlaute) vorschlagen.

Vokalbuchstabe	Ersetzungen
A	4 , @ , /\ , /-\ , ^ , α , λ
E	3 , € , £ , ε
I	! , 1 , | ,][, ǐ
O	0 , () , [] , * , ° , <> , ø , {[]}
U	|_| , µ , [_] , v
Y	`/ , °/ , ¥

Tabelle 14.2 *Leet-Speak*

Unter Beachtung dieser Ersetzungen würde man also aus der ersten Variante `EP-Emj,aelD.` das relativ sichere Kennwort `3P-Emj,4elD.` erhalten. Allerdings wäre hierbei die Zeichendiversifikation noch optimierbar, indem man eine der anderen Ersetzungen wählt.

Das fertige Regelset

Versuchen wir nun, unsere Erkenntnisse in einer ***Validator-Klasse*** zu berücksichtigen. Der folgende Ansatz orientiert sich an »Sichere Webanwendungen mit PHP« von Wassermann S. 111f. (siehe Abschnitt 15.3).

Beispiel

```
1 <?php
2
3 namespace Validators;
4
5 use Webmasters\Doctrine\ORM\EntityValidator;
6
```

```php
class UserValidator extends EntityValidator
{
    // Mindestlaenge des Kennworts
    protected $minLength = 12;

    public function validatePassword($password)
    {
        $entity = $this->getEntity(); // komplettes Benutzer

        // Ermittlung aller benutzten Zeichen
        $usedChars = count_chars($password, 1);

        // Es kommt min. ein Buchstabe A-Z vor
        $hasLetters = $this->filterRegex($password, '/[A-Z]+

        // Es kommt min. ein Buchstabe a-z vor
        $hasSmallLetters = $this->filterRegex($password, '/[

        // Es kommt min. eine Zahl vor
        $hasNumbers = $this->filterRegex($password, '/\d+/')

        // Es kommt min. ein Sonderzeichen vor
        $hasSpecialChars = $this->filterRegex($password, '/[

        if (empty($password)) {
            $this->addError('Das Feld Passwort ist leer.');
        } elseif (strlen($password) < $this->minLength) {
            $this->addError(
                sprintf('Das Passwort sollte mindestens %d Ze
            );
        } elseif (count($usedChars) < (strlen($password) / 2
            $this->addError('Das Passwort sollte zu mindeste
        } elseif (
            ($hasLetters === false) ||
            ($hasSmallLetters === false) ||
            ($hasNumbers === false) ||
            ($hasSpecialChars === false)
        ) {
            $this->addError('Das Passwort sollte Großbuchsta
        } elseif (
            $entity->getEmail() &&
            (stristr($password, $entity->getEmail()) !== fal
        ) {
            $this->addError('Das Passwort sollte keine priva
        }
    }

    protected function filterRegex($wert, $regex)
    {
        return filter_var(
```

```
57                  $wert,
58                  FILTER_VALIDATE_REGEXP,
59                  array(
60                      'options' => array('regexp' => $regex)
61                  )
62              );
63      }
64  }
```

src/Validators/UserValidator.php

Codebeispiel 160 *src/Validators/UserValidator.php*

Diese Klasse ist nicht wirklich vollständig, zeigt aber gut, wohin die Reise geht. Vor allem hakt es noch bei der Überprüfung der persönlichen Daten, da uns im Beispiel des Newstickers lediglich die E-Mail-Adresse vorliegt. Doch gehen wir kurz die Besonderheiten des Codes durch. In Zeile 10 legen wir die Mindestlänge von Kennwörtern als Attribut unserer Klasse fest. In Zeile 54-63 definieren wir eine neue Methode `UserValidator#filterRegex()`, die lediglich einen verkürzten Zugriff auf die Funktion filter_var() bietet. Diese Funktion existiert seit PHP 5.2.0 und bietet neben der verwendeten Alternative für `preg_match()` auch Validierungsfilter für E-Mail-Adressen und URLs. Doch dies ist noch nicht alles. Die Validierungsfilter sind jedoch teilweise etwas mit Vorsicht zu genießen, da die erlaubten Werte nicht unbedingt zwingend den gewünschten Werten entsprechen, wie man an den Kommentaren auf php.net erkennt. Ich habe mich übrigens in diesem Beispiel gegen `preg_match()` entschieden, da diese Funktion einen nicht gefundenen regulären Ausdruck durch den Integer-Wert `0` signalisiert. Die Funktion `filter_var()` verwendet hierfür hingegen den lesbareren Boolean `false`.

Die letzte Besonderheit sind die Zeilen 17 und 37. In Zeile 17 ermitteln wir zunächst mittels count_chars() ein Array aller verwendeten Zeichen. In Zeile 37 prüfen wir dann, ob die Anzahl der Array-Elemente weniger als die halbe Stringlänge des Kennwort-Werts beträgt. Wäre dies der Fall, so läge ein Verstoß gegen die 50%-Regel der **Zeichendiversifikation** vor.

Übung 51:

1. Wir benötigen einen neuen Link `Register` im Partial *navi.tpl.php*. Dieser soll zu der neuen Methode `UserController#registerAction()` führen.

2. Setzen Sie die Registrierungs-Aktion inklusive des Templates *templates/UserController/editAction.tpl.php* um. Die Kennwörter sollen derzeit so in der Datenbank gespeichert werden, wie sie auch eingegeben wurden (sogenannter Klartext). Implementieren Sie außerdem die vorgestellte Klasse `UserValidator` in Ihrem Projekt.
3. Testen Sie das Regelset mit den nachfolgenden Kennwörten: `geheim`, `1111111111`, `1234567890`, `Andrea1984` und `3P-Emj,4elD`.
4. Ergänzen Sie abschließend noch Validierungen für die E-Mail-Adresse (z. B. keine doppelten E-Mail-Adressen).

Es ist wenig komfortabel, wenn wir den Sicherheitsstatus immer erst nach dem Absenden des Formulars anzeigen und auch nur das erste Validierungsproblem des Passworts melden. Bei derart hohen Anforderungen würden die meisten Benutzer nach ein paar Fehlschlägen entnervt aufgeben und auf eine Registrierung verzichten. Geben Sie also immer vorab im Formular entsprechende Hinweise oder nutzen Sie JavaScript, um den Status schon während der Eingabe anzuzeigen. Hierbei wäre es auch denkbar, unsere PHP-Validierungen mittels Ajax anzusprechen, um identische Regeln zu verwenden und diese nicht in zwei Programmiersprachen implementieren zu müssen.

14.6.2 Passworthashing

Wie der Fall von RockYou.com zeigt, ist Klartext nicht die empfehlenswerteste Variante, um wichtige Daten (z. B. Kennwörter oder Kreditkarteninformationen) zu speichern. Solche Daten müssen zwar für uns und unsere Anwendung verfügbar sein, aber möglichst für keinen Dritten. Dies erreichen wir, indem wir die Klartextwerte verstecken. Zunächst müssen wir jedoch erst einmal verstehen, dass es hierfür zwei fundamental unterschiedliche Ansätze gibt.

1. **Verschlüsselung**: Die Information wird mit einem Verschlüsselungs-Algorithmus behandelt. Dieser ist unter Zuhilfenahme eines Schlüssels (am sichersten sind sehr lange zufällige Werte) in der Lage, den Klartext in eine chiffrierte Form umzuwandeln und dies auch wieder rückgängig zu machen. Eine Verschlüsselung dient oftmals dazu, Informationen zwischen zwei Personen oder Gruppen auszutauschen. Es gibt **symmetrische** und

asymmetrische Verfahren. Bei einem symmetrischen Verfahren müssen beide Gruppen Kenntnis über einen einzigen gemeinsamen Schlüssel haben. Beim asymmetrischen Verfahren wird mit einem geheimen privaten und einem öffentlichen Schlüssel gearbeitet. Die Daten werden dann beispielsweise mit dem öffentlichen Schlüssel chiffriert und können nur mit dem privaten Schlüssel wieder dekodiert werden.

2. **Hashing**: Die Information wird mit einem Einweg-Hash-Algorithmus (engl. One-Way Hash Function) behandelt. Der Klartextwert selbst dient als Schlüssel. Aus einem Klartextwert mit beliebiger Länge entsteht ein sogenannter Hash (engl. to hash, dt. zerhacken) mit im Normalfall fester Länge. Der Vorgang selbst ist nicht umkehrbar. Das Verfahren basiert auf der Annahme, dass der entstandene Hash eine einzigartige Signatur (engl. Fingerprint) des ursprünglichen Wertes ist und den Wert so eindeutig identifiziert, ohne etwas über seinen Inhalt zu verraten. Um einen Wert anhand eines Hashs zu überprüfen, muss lediglich der zu prüfende Wert mit dem gleichen Verfahren gehasht und dann die beiden Signaturen verglichen werden.

Für Kennwörter ist das Hashing empfehlenswerter, da für eine Verifizierung (engl. Error Detection) des Passworts ein Vergleich der Signaturen ausreicht. Deswegen werden wir uns diese Variante einmal genauer ansehen. Doch welche Hashing-Algorithmen gibt es und was sollten wir bei der Auswahl bedenken?

Ein guter und sicherer Einweg-Hash-Algorithmus sollte unbedingt ein paar wichtige Eigenschaften erfüllen:

1. **Identische Ergebnisse**: Der gleiche Algorithmus muss aus einem bestimmten Klartextwert immer den identischen Hash erzeugen.
2. **Chaotische Ergebnisse**: Ähnliche Klartextwerte müssen selbst bei nur einem sich unterscheidenden Zeichen zu völlig unterschiedlichen Hashwerten führen.
3. **Einwegfunktion**: Es muss praktisch unmöglich sein, aus einem Hash den zugehörigen Klartextwert zu ermitteln.
4. **Starke Kollisionsresistenz**: Es darf zwar theoretisch möglich sein, dass zwei Klartextwerte den gleichen Hash erzeugen (sogenannte Kollisionen).

Es muss jedoch praktisch unmöglich sein, für einen bestimmten Hash einen weiteren Klartextwert zu ermitteln, der eine Kollision erzeugt (dt. kollisionsresistente Hashfunktion, engl. Collision Resistant Hash Function).

Die bekanntesten und gleichzeitig für Kennwörter nicht empfehlenswerten Algorithmen sind:

- **Message Digest**: Es handelt sich hierbei um eine Reihe von Hashfunktionen, die von *Ronald L. Rivest* am *Massachusetts Institute of Technology* entwickelt wurden. MD5 (also die 5. Version) ist derzeit noch weit verbreitet, aber bei kurzen Klartextwerten in minimaler Zeit mittels Brute-Force-Methode angreifbar.
- **Secure Hash Algorithm**: Der ursprüngliche Algorithmus wurde in einer Zusammenarbeit von NIST (*National Institute of Standards and Technology*) und NSA entwickelt. Hieraus entstand zunächst die fehlerbereinigte Fassung SHA-1. Als Reaktion auf bekannt gewordene Angriffe gegen SHA-1 entstand die SHA-2-Familie (SHA-256, SHA-384, SHA-512 und SHA-224).

Auch wenn SHA-2 laut NIST noch als sicher gilt, haben alle genannten Verfahren einen gravierenden Nachteil, da sie mit dem Ziel entwickelt worden sind, auch größere Datenmengen möglichst effizient zu hashen. Diese Effizienz und hierbei vor allem der Faktor Zeit erleichtert es aber Angreifern, die Passwörter mittels Brute-Force-Attacken zu ermitteln. Aus diesem Grund wurde *bcrypt* speziell für das Hashing von Kennwörtern entwickelt. Bcrypt verfügt über einen je nach verfügbarer Serverhardware einstellbaren Kosten- bzw. Zeitfaktor. Mit diesem Faktor kann später auch der Aufwand erhöht werden, wenn sich die Leistungsfähigkeit der Computer weiterentwickelt hat. Bei *bcrypt* handelt es sich jedoch um keinen eigenen Hashing-Algorithmus, sondern um eine kryptologische Hashfunktion, die intern Teile des Verschlüsselungs-Algorithmus *Blowfish* zur Erzeugung eines Einweg-Hashes verwendet.

> Legen Sie Kennwörter stets als ***Hash*** in einer Datenbank ab, sofern diese lediglich zur Login-Prüfung benötigt werden.

Passwort-Hashing-API

Anthony Ferrara hat am 26.06.2012 ein RFC (engl. Request for Comments, dt. Bitte um Kommentare) für eine simple Passwort-Hashing-API auf Basis von *bcrypt* eingereicht, die ab **PHP 5.5** im Core von PHP implementiert wurde.

Die API besteht aus vier Funktionen, die ich im Folgenden am Beispiel von Mini-Controllern kurz vorstellen möchte. Denken Sie jedoch bitte daran, für diese Tests die *.htaccess* zu entfernen.

Beispiel

```php
<?php

$password = 'test'; // Benutzereingabe
$hash = password_hash($password, PASSWORD_DEFAULT);

var_dump($hash);
```

api_version_1.php - password_hash()

Codebeispiel 161 *api_version_1.php - password_hash()*

Zum Erzeugen eines Hashwertes dient die Funktion password_hash(), die als ersten Parameter den Klartextwert des Kennworts erhält. Als zweiten Parameter teilen wir den gewünschten Algorithmus mit. Die Konstante `PASSWORD_DEFAULT` steht hierbei für den aktuell als am sichersten betrachteten Algorithmus, was derzeit bcrypt ist. Mit zukünftigen PHP-Versionen kann sich dies jedoch ändern. Wenn Sie übrigens keine automatische Anpassung des Algorithmus im Rahmen von neuen PHP-Versionen wünschen, so können Sie stattdessen die Konstante `PASSWORD_BCRYPT` benutzen.

Beispiel

```php
<?php

$password = 'test'; // Benutzereingabe
$hash = password_hash($password, PASSWORD_DEFAULT);

$info = password_get_info($hash);
var_dump($hash, $info);
```

api_version_2.php - password_get_info()

Codebeispiel 162 *api_version_2.php - password_get_info()*

Betrachten wir nun die Informationen zu unserem Hash, welche wir durch die Funktion `password_get_info()` erhalten. Bei Erstellung dieses Kapitels wurde mir beispielsweise mitgeteilt, dass der Algorithmus `bcrypt` und der Kostenfaktor `10` benutzt wurden. Beide Informationen kann man auch direkt dem Beginn des Hashwertes `$2y$10$` entnehmen. Das `2y` steht für die seit PHP 5.3.7 verfügbare, fehlerbereinigte Implementierung von bcrypt und die `10` für den Kostenfaktor. Für den Kostenfaktor sind Werte zwischen 4 und 31 möglich. Je nach verwendeter Serverhardware sollten Sie eine Erhöhung des Wertes (> 10) erwägen. Ein Aufwand von 0,5 Sekunden für die Hashermittlung sollte laut *Anthony Ferrara* jedoch die obere zeitliche Grenze bilden.

Wenn Sie den Mini-Controller mehrfach aufrufen, so sollte Ihnen auffallen, dass Sie immer einen anderen Hash erhalten (mit Ausnahme der Informationen am Beginn). Dies ist kein Verstoß gegen die Regel der identischen Ergebnisse und liegt lediglich daran, dass man der Funktion `password_hash()` neben dem Kostenfaktor auch noch einen zweiten Optionsparameter mitteilen kann. Hierbei handelt es sich um einen sogenannten ***Salt*** (dt. Salz). Ein sinnvoller Salt ist für jeden Benutzer anders und dient dazu, dass man aus zwei identischen Hashwerten in einer Datenbank nicht auf ein identisches Passwort im Klartext folgern kann. Ein solcher Salt muss übrigens nicht geheim sein, sondern kann zusammen mit dem Kennwort in der Datenbank abgelegt werden.

Beispiel

```php
<?php

$password = 'test'; // Benutzereingabe
$options = array(
    'cost' => 12,
    'salt' => '12345678901234567890123', // min. 22 Zeichen "
);

$hash = password_hash($password, PASSWORD_DEFAULT, $options)

$info = password_get_info($hash);
var_dump($hash, $info);
```

api_version_3.php - password_hash() mit Optionen

Codebeispiel 163 *api_version_3.php - password_hash() mit Optionen*

Wenn Sie diesen Code mehrfach ausführen lassen, so sollten drei Dinge

auffallen:

1. Der Salt taucht **nicht** unter `options` bei `password_get_info()` auf.
2. Der Hash beginnt mit `$2y$12$123456789012345678901`. Dies liegt daran, dass wir nun einen Kostenfaktor von `12` und einen selbst festgelegten und leicht zu erkennenden Salt (dritte Angabe im Hash) verwenden.
3. Es werden nur 21 Stellen unseres Salts im Hash angezeigt. Dies liegt daran, dass der Salt von bcrypt grundsätzlich aus 21 und einem zerquetschten Zeichen besteht. Gibt man weniger als 22 Zeichen an, so erhält man die PHP-Warnung `Provided salt is too short`. Gibt man hingegen mehr als 22 Zeichen an, so werden die überzähligen Zeichen **ignoriert**.

Machen wir nun einen kleinen Test und verwenden hierfür den kleinsten erlaubten Kostenfaktor, also `4`.

Beispiel

```php
<?php

$password = 'p';
$options = array(
    'cost' => 4,
    'salt' => '12345678901234567890123',
);

$hashes = array();
for ($i = 1; $i < 100; $i++) {
    $hash = password_hash($password, PASSWORD_DEFAULT, $optio

    if (isset($hashes[$hash])) {
        break;
    }

    $hashes[$hash] = $i;
    $password .= 'p';
}

?>

<pre><?php var_dump($hashes); ?></pre>
<p><?php echo $i; ?>: <?php echo $hash; ?></p>
```

api_version_3c.php

Codebeispiel 164 *api_version_3c.php*

Was macht dieser Code? In Zeile 3 wird als Passwort genau ein Buchstabe festgelegt. Für dieses Passwort wird in der `for`-Schleife ein Hash erzeugt und in einem Array abgelegt, sofern er dort nicht existiert. Hierbei wird der Hash als Array-Schlüssel und die Zählvariable als dazugehöriger Wert benutzt, da `isset()` wesentlich performanter als `in_array()` ist. Bei jedem weiteren Schleifendurchlauf wird für das Kennwort ein Zeichen mehr verwendet. Nach dem Abarbeiten der Schleife wird das Array ausgegeben. Außerdem wird in Zeile 24 der letzte Inhalt der Zählvariable und der dazugehörige Hash ausgegeben.

Wenn Sie den Mini-Controller nun im Browser ausführen, so fällt als erstes die Geschwindigkeit auf, mit der das Hash-Array generiert wird. Verwenden Sie also bitte **niemals** in einem realen Projekt einen so geringen Kostenfaktor. Wenn Sie nun das Ende der Ausgabe genauer betrachten, so sollte noch etwas auffallen. Der letzte Array-Wert (bei mir `72`) hat den **gleichen Hash**, der auch im nachfolgenden (also 73.) Durchlauf erzeugt wurde.

> Bei bcrypt ist die unterstützte Zeichenanzahl begrenzt. Abhängig von der Implementierung werden ab einer bestimmten Grenze die nachfolgenden Zeichen bei der Erzeugung eines Hashes ignoriert. Alle mir bekannten Implementierungen unterstützen jedoch mindestens 50 Zeichen. Dieses Problem kann man jedoch meiner Meinung nach ignorieren, da ja trotzdem ein gültiger Hash erzeugt wird. Wir haben lediglich keinen weiter erhöhten Schutz gegen Brute-Force-Attacken, wenn wir die Grenze überschreiten.

Persönlich sehe ich übrigens auch keinen Grund, den Salt manuell anzugeben, und verzichte deswegen im Weiteren darauf.

Beispiel

```
1 <?php
2
3 $password = 'test'; // Benutzereingabe
4 $hash = '$2y$12$kIFAPDlY.N06arvd7IKZN' .
```

```
    'O7kXop6EngNgGz19yptFJyWGOoMjqSf.';

if (password_verify($password, $hash)) {
    $meldung = 'Passwort ist korrekt';
} else {
    $meldung = 'Passwort ist falsch';
}

echo $meldung;
```

api_version_4.php - password_verify()

Codebeispiel 165 *api_version_4.php - password_verify()*

Um ein Kennwort im Klartext mit einem Hash zu vergleichen, benötigen wir die Funktion `password_verify()`. Diese Funktion benötigt lediglich die zu verifizierende Passworteingabe und den Hash als Parameter, da die verwendeten Optionen ja anhand des Hashbeginns ersichtlich sind.

Beispiel

```
<?php

$password = 'test'; // Benutzereingabe
$options = array(
    'cost' => 15,
);

$hash = '$2y$12$kIFAPDlY.N06arvd7IKZN' .
    'O7kXop6EngNgGz19yptFJyWGOoMjqSf.';

if (password_verify($password, $hash)) {
    echo 'Bisheriger Hash: ' . $hash;

    // Kostenfaktor wurde von 12 auf 15 erhoeht
    if (password_needs_rehash($hash, PASSWORD_DEFAULT, $optio
        // Neuen Hash mit dem aktuellen Kostenfaktor ermittel
        $hash = password_hash($password, PASSWORD_DEFAULT, $o

        /* Hier fehlt die Aktualisierung des Hashwertes in de
        echo ' Neuer Hash: ' . $hash;
    }

}
```

api_version_5.php - password_needs_rehash()

Codebeispiel 166 *api_version_5.php - password_needs_rehash()*

Persönlich neige ich dazu, für den Kostenfaktor wegen der Abhängigkeit von

der verwendeten Hardware einen eigenen Wert und als Algorithmus `PASSWORD_DEFAULT` anzugeben. Dies verhindert zwar eine automatische Nutzung eines im Rahmen von PHP-Updates erhöhten Standard-Faktors und legt diese Anpassung in die Hand des Entwicklers, ermöglicht aber schon jetzt einen besser geschützten Hashwert. Eine erneute Entscheidung bezüglich der benutzten Parameter sollte übrigens regelmäßig (beispielsweise nach jedem Wechsel der Serverhardware) gefällt werden. Das obige Beispiel verdeutlicht, wie im Rahmen einer Loginprüfung mittels der Funktion `password_needs_rehash()` ermittelt werden kann, ob nach einer Parameteranpassung ein Hash in der Datenbank aktualisiert werden muss.

> Wenn Sie den Standard-Algorithmus verwenden, sollten Sie unbedingt berücksichtigen, dass der Hash von bcrypt zwar nur 60 Zeichen benötigt, dies aber nicht zwingend für zukünftige Alternativen gilt. Erlauben Sie deshalb schon jetzt in der Datenbank **255 Zeichen für den Hash** oder verwenden Sie die Konstante `PASSWORD_BCRYPT`.

14.6.3 Benutzernamen und Kennungen

Wir haben nun also gelernt, wie wir relativ sichere Kenwörter erzwingen und diese mit der Passwort-Hashing-API verwenden. Ein Login ist jedoch um ein vielfaches sicherer, wenn er aus zwei schwer zu ermittelnden Komponenten besteht. Schenken Sie einem Angreifer also nicht einfach eine dieser Komponenten, indem Sie den Benutzernamen (bei uns die E-Mail-Adresse) öffentlich anzeigen. Beispielsweise könnten Sie Ihre `User`-Entity um ein Attribut `$displayName` erweitern, das Sie dann für die öffentliche Anzeige des Erstellers eines `Article` benutzen.

Wenn Sie in einer Anwendung eine Loginprüfung umsetzen, so sollten erfolglose Anmeldeversuche immer nur mit einer kurzen Fehlermeldung **ohne Angabe von Einzelheiten** abgelehnt werden. Insbesondere darf in einem solchen Fall nicht erkennbar sein, ob der eingegebene Benutzername oder das Kennwort (oder beides) falsch ist. Unterlassen Sie die Umsetzung dieser Empfehlung, so kann ein Angreifer mit einem Wörterbuchangriff zunächst gültige Benutzernamen ermitteln, und das ist schon die halbe Miete. Benutzen bzw. erlauben Sie zudem

niemals typische Benutzernamen wie **admin**, **administrator**, **root** oder **superuser**. Bei automatisierten Angriffen werden diese immer zuerst ausprobiert.

14.6.4 Browserfeatures

Seit etlichen Jahren verfügen die bekannten Browser (IE, Firefox, Chrome, Safari und Opera) über Features, die den Benutzer beim Besuch von Websites unterstützen sollen. Doch auch aus diesen eigentlich hilfreichen Features können sich schnell Probleme für uns als Entwickler ergeben. Ich möchte im Folgenden zwei dieser Features am Beispiel des Browsers *Firefox* vorstellen und Ihnen zeigen, wie Sie diese zu Lasten des Benutzerkomforts deaktiveren können.

Formular-Autovervollständigung

Firefox merkt sich, welche Daten man in einzeilige Textfelder eingegeben hat. Nach einer solchen Eingabe ist diese beim nächsten Besuch der Webseite wieder verfügbar und wird nach der Angabe des ersten Buchstabens als Vorschlag angezeigt.

Dieses Feature hört sich zunächst einmal toll an. Doch was ist, wenn Ihr Benutzer einmal kurz seinen Rechner verlassen muss und in dieser Zeitspanne ein Unbefugter Zugriff auf den Browser hat? Der Angreifer bräuchte lediglich das erste Zeichen auszuprobieren und bekäme sofort den kompletten Benutzernamen serviert.

Beispiel

```
<form action="index.php?controller=user&amp;action=login" me

    <label for="email">E-Mail</label>
    <input type="text" name="email" id="email" />

    <label for="password">Password</label>
    <input type="password" name="password" id="password" />

    <input type="submit" class="button" value="Anmelden" />

```

```
11 </form>
```
templates/UserController/loginAction.tpl.php (Version 1)

***Codebeispiel 167** templates/UserController/loginAction.tpl.php (Version 1)*

Das Kennwort ist hiervon übrigens nicht betroffen, da es für `password`-Felder keine Vorschlagsfunktionalität gibt. Trotzdem besteht auch eine Gefahr für dieses. Erst vor kurzem hatte ich in meiner eigenen Familie den Fall, dass eine Login-Eingabe zu schnell erfolgte und dabei die Tabulator-Taste nicht korrekt aktiviert wurde. Hierdurch landete eine kombinierte Eingabe von Benutzername und Passwort in den automatischen Vorschlägen für den Benutzernamen. Dank der anderen Vorschläge wäre es dann ein Leichtes gewesen, den Beginn des Kennworts zu identifizieren.

Passwort-Manager

Der Passwort-Manager von Firefox speichert die Benutzernamen und Kennwörter, die man auf Webseiten verwendet, und fügt sie beim nächsten Besuch automatisch in das Login-Formular ein. Hierbei kann man als Benutzer entscheiden, ob eine Speicherung erfolgen soll oder nicht.

Dieses Feature kann gleich in mehreren Szenarien nachteilig sein:

1. **Ein Cross-Site-Scripting-Angriff auf das Login-Formular**: Ein Angreifer muss nicht erst auf die Eingabe der Daten durch den Benutzer warten, sondern kann diese sofort übermitteln.
2. **Ein Unbefugter mit direktem Browser-Zugriff**: Der Angreifer sieht sofort den Benutzernamen im Formular. Zudem kann er sich bei ausreichend Zeit einloggen und das Passwort ändern. Eine Anwendung sollte deswegen vor einer solchen Änderung als **zusätzliche Verteidigungslinie** immer nach dem bisherigen Kennwort fragen. Wenn der Angreifer jedoch genug Zeit hat, könnte er auch eine Browser-Erweiterung installieren, um vorübergehend die Sternchen im Passwortfeld zu demaskieren.
3. **Ein Unbefugter, der die Festplatte (bzw. den kompletten Rechner) entwendet oder Daten kopiert**: Obwohl der Passwort-Manager die Zugangsdaten in einem verschlüsselten Format auf der Festplatte speichert, kann sich ein Angreifer mit ausreichend Zeit und Rechenpower Zugriff auf die Daten im Klartext verschaffen. Eine Speicherung als Hash ist

nämlich nicht möglich, da die Klartextwerte für die Zuweisung in die jeweiligen Felder des Login-Formulars benötigt werden. Ich hoffe, Sie verwenden zumindest ein langes und komplexes **Masterkennwort** (sofern Sie selbst einen Passwort-Manager verwenden), um so die benötigte Angriffszeit zu erhöhen.

Um diese Probleme zu vermeiden, wurde seit dem Internet Explorer 5 (Firefox ab 0.9.4) die Unterstützung des nicht standardisierten Attributs autocomplete in die Browser implementiert. Es wird in den aktuellen Versionen von allen bekannten Browsern unterstützt und ist Teil des HTML5-Standards.

Beispiel

```
<form
    action="index.php?controller=user&amp;action=login" metho
    autocomplete="off"
>

    <label for="email">E-Mail</label>
    <input type="text" name="email" id="email" />

    <label for="password">Password</label>
    <input type="password" name="password" id="password" />

    <input type="submit" class="button" value="Anmelden" />

</form>
```

templates/UserController/loginAction.tpl.php (Version 2)

Codebeispiel 168 *templates/UserController/loginAction.tpl.php (Version 2)*

Das Attribut kann im öffnenden `form`-Tag ergänzt werden und deaktiviert so die Autovervollständigung für alle Felder dieses Formulars.

Beispiel

```
<form action="index.php?controller=user&amp;action=login" me

    <label for="email">E-Mail</label>
    <input
        type="text" name="email" id="email"
        autocomplete="off"
    />

```

```
 9      <label for="password">Password</label>
10      <input
11          type="password" name="password" id="password"
12          autocomplete="off"
13      />
14
15      <input type="submit" class="button" value="Anmelden" />
16
17  </form>
```

templates/UserController/loginAction.tpl.php (Version 2b)

Codebeispiel 169 *templates/UserController/loginAction.tpl.php (Version 2b)*

Alternativ ist auch der Einsatz in einzelnen Formularfeldern möglich, was ich persönlich bevorzuge.

> Das Attribut `autocomplete` funktioniert mit folgenden Typen des `input`-Tags: text, password, search, url, tel, email, range, color und den Datumstypen.

Wichtig ist jedoch, dass das Attribut lediglich als Empfehlung an den Browser angesehen werden kann. Da es sich um eine Angabe im HTML-Quelltext einer Webseite handelt, kann ein Benutzer diese »Gängelung durch den Seitenbetreiber« beispielsweise mittels Browser-Erweiterung umgehen.

14.7 Benutzereingaben

Wenn man beginnt, sich mit der Sicherheit eigener Anwendungen zu beschäftigen, so erhält man viele gut gemeinte Ratschläge im Zusammenhang mit den Eingaben von Benutzern. Dazu gehören beispielsweise:

1. **Never trust user input!**
2. **Always sanitize your inputs!**

Den ersten Punkt habe ich in diesem Lernbuch mit **»Traue niemals deinen Benutzern und ihren Eingaben«** übersetzt und direkt zu Anfang dieser Lektion behandelt. Doch wie steht es mit dem zweiten Punkt? Versuchen wir hierzu eine Differenzierung der drei englischen Begriffe ***Validating***, ***Sanitizing*** und ***Escaping***.

- **Validating/Validierung:** Die Anwendung überprüft, ob eine Benutzereingabe den Erwartungen entspricht. Ist ein Wert gültig, so kann er weiterverarbeitet (z. B. gespeichert) werden. Ist dies nicht der Fall, so wird der Wert abgelehnt.

- **Sanitizing/Filterung:** Die Anwendung verändert die Benutzereingaben und entfernt beispielsweise nicht erwünschte Teile im Wert (z. B. Einheiten am Ende einer Zahl oder HTML-Tags in Texten). Normalerweise können die ursprünglichen Werte nicht wiederhergestellt werden, da ja etwas wegfällt. Ein gefilterter Wert ist aber nicht zwingend valide!

- **Escaping/Maskierung:** Die Anwendung wandelt die Benutzereingabe um, damit wir beim gewünschten Ausgabemedium von Sicherheitsproblemen verschont bleiben. Für HTML verwendet man meist die Funktion `htmlspecialchars` (also eine Zeichenersetzung). Es gibt jedoch nicht nur das Ausgabemedium HTML, sondern beispielsweise auch CSV, PDF, Grafik und JSON. Für jedes dieser Medien kann ein ganz anderes Escaping nötig sein.

> **Achtung**
>
> Der Begriff ***Sanitizing*** ist nicht eindeutig belegt, was bei der Kommunikation zwischen Programmierern leider oftmals zu Missverständnissen führt. Teilweise wird das ***Escaping*** als Teilbereich dieser Filterung gesehen und man redet von ***Sanitizing***, obwohl man ***Escaping*** meint. Unser zweiter Ratschlag resultiert genau hieraus. Korrekt wäre aus meiner Sicht somit eher: **»Führe immer ein Escaping von Benutzereingaben durch!«**

Wann ist nun was zu empfehlen?

Bei der serverseitigen **Validierung** mit PHP ist dies relativ einfach zu beantworten. Diese sollte **vor einer Speicherung** erfolgen. Hierbei ist auch nicht relevant, wo die Speicherung erfolgt. Dies könnte beispielsweise in einer Datenbank, in einer Datei oder einfach in der Session sein. Für manche Werte bieten sich hierbei die Validate-Filter von `filter_var` an. Allerdings sind diese auch nicht unproblematisch, da sie teilweise zu viele Eingaben erlauben oder

theoretisch gültige Eingaben (z. B. asiatische Zeichen) ablehnen.

Beispiel

```
public function readAction()
{
    $em = $this->getEntityManager();
    $article = $em
        ->getRepository('Entities\Article')
        ->find((int)$_GET['id'])
    ;

    $article || $this->render404();

    $this->addContext('article', $article);
}
```

URL-Parameter

***Codebeispiel 170** URL-Parameter*

Die Anweisung `$article || $this->render404()` ist auch eine Art von **Validierung**. Allerdings prüfen wir hiermit, ob ein korrekter Wert durch Verwendung eines URL-Parameters ermittelt werden konnte. Ist dies der Fall, so arbeiten wir weiter mit dem Wert. Ist dies nicht der Fall, so zeigen wir eine 404-Fehlermeldung an. Eine solche Überprüfung sollte also **vor einer Weiterverarbeitung** des Wertes erfolgen.

Escaping sollte grundsätzlich **im Rahmen der Ausgabe** des Wertes stattfinden. Theoretisch wäre es natürlich denkbar, sich einen maskierten Wert vorab (z. B. in einer Variable oder gar einer Datenbank) zu merken und diesen dann mehrfach zu verwenden. Dies würde aber die Lesbarkeit des Codes negativ beeinflussen, da man so bei der Ausgabe nicht mehr sofort nachvollziehen könnte, ob ein Escaping erfolgt ist und dies wiederum zum Unterlassen eines Escapings führen könnte. Außerdem kann es sein, daß man je nach Stelle in der Ausgabe (z. B. ein Attribut oder sogar ein href-Attribut) unterschiedliche Methoden des Escapings verwenden möchte. Dies ist unmöglich oder wird zumindest erschwert, wenn bereits vorab maskiert wurde.

Doch wie sieht es nun mit dem **Sanitizing** aus? Gehen wir von einer Zahl mit Einheit aus und könnten wir in unserer Datenbank nur eine Zahl speichern, so sollten wir natürlich vor dem Speichern filtern. Dies könnte **beispielsweise im**

Setter erfolgen. Unser Attribut würde dann immer eine Zahl beinhalten, welche wir dann validieren und danach speichern können. Ein Sanitizing kann also eine **Ergänzung zur Validierung** sein, um die Usability der Anwendung zu erhöhen. Persönlich meide ich diese Variante etwas und lehne bei Formularen ungültige Werte lieber komplett ab. Aber das ist definitiv Geschmackssache.

Ein anderer Anwendungsfall von **Sanitizing** ist die Bereinigung von URL-Parametern. Ein `(int)$_GET['id']` ist eine Filterung, da die ID danach nur noch eine Zahl sein kann. In Codebeispiel 170 handelt es sich hierbei ebenfalls um eine **Ergänzung zur Validierung** (in Form von Zeile 9). Die Sanitize-Filter von `filter_var` bzw. `filter_input` fallen teilweise in diesen Bereich (manche machen eher ein Escaping).

Der HTML-Purifier und `strip_tags` fallen meiner Meinung nach in den Bereich des **Sanitizing**. Wann bzw. wo sollte man eine solche Filterung einsetzen? Eine Filterung **vor der Speicherung** ist aus meiner Sicht für viele Anwendungen ungünstig bzw. **nicht zu empfehlen**. Beispielsweise könnte es zwei Benutzerrollen – Admin und Anwender – geben, und Admins sollen Eingaben inklusive HTML-Formatierungen sehen können, Anwender nicht. Man hat aber erst im Rahmen der Ausgabe die Information, welcher Benutzer mit welcher Rolle den Wert sehen will. Würde man vor dem Speichern bereinigen, wären die nötigen Informationen im Wert schon zerstört bzw. nicht mehr vorhanden. In diesem Fall sollte das Sanitizing also **als Ersatz bzw. Ergänzung des Escapings bei der Ausgabe** verwendet werden.

Eine Filterung tritt also in zwei möglichen Konstellationen auf:

- **Sanitizing** als Ergänzung (oder ggf. auch Ersatz) zur Validierung:
  Diese Variante sollte in einer Anwendung **sehr früh** erfolgen. Der entsprechende Code sollte sich natürlich über der Validierung (z. B. ziemlich am Anfang der Controller-Aktion) bzw. zeitlich davor befinden (z. B. im Setter).

- **Sanitizing** als Ergänzung oder Ersatz des Escapings:
  Diese Variante sollte in einer Anwendung **sehr spät** erfolgen. Eine mögliche Umsetzung ist hierbei die direkte Kombination mit der Ausgabe, wie wir es in der `e`-Funktion machen.

Übung 52:

Testen Sie die Registrierung des Newstickers ohne Angabe einer E-Mail-Adresse und mit dem Kennwort `0123abc<DEF>`. Was passiert, und was passiert nach einem sofortigen erneuten Abschicken des Formulars?

Unsere ***Validierung*** legt für Eingaben im Password-Feld eine Mischung von Zeichen fest, zu denen auch Sonderzeichen gehören sollen. Die Zeichen < und > sind solche Sonderzeichen und somit theoretisch im Kennwort erwünscht. Dummerweise bereitet unser ***Sanitizing*** im Falle eines Validierungsfehlers Probleme. Wenn man genau darüber nachdenkt, so widerspricht das generelle ***Sanitizing*** mit `strip_tags` in der `e`-Funktion im Falle von Formularfeldern meiner Empfehlung, mit `strip_tags` nicht vor dem Speichern zu filtern.

Beispiel

```php
function e($dirty, $stripTags = true, $encoding = 'UTF-8')
{
    if ($stripTags) {
        $dirty = strip_tags($dirty);
    }

    echo htmlspecialchars(
        $dirty,
        ENT_QUOTES | ENT_HTML5,
        $encoding
    );
}
```

Ausschnitt helper.inc.php (e-Funktion V2)

Codebeispiel 171 *Ausschnitt helper.inc.php (e-Funktion V2)*

```php
<label for="password">Password*</label>
<input
    name="password" id="password" type="password" maxlength="
    value="<?php e($user->getPassword(), false); ?>"
/>
```

Ausschnitt UserController\editAction.tpl.php

Codebeispiel 172 *Ausschnitt UserController\editAction.tpl.php*

Vergleicht man diese neue Version der `e`-Funktion mit unserem bisherigen Stand, so fällt als Erstes der neue Parameter `$stripTags` auf. Dieser ist

standardmäßig `true`. Wenn wir nun beim Funktionsaufruf als Wert für diesen zweiten Parameter `false` angeben, so findet kein ***Sanitizing*** statt. Für die Ausgabe von Werten in Formularfeldern verwenden wir fortan **immer** die `e`-Funktion mit deaktiviertem ***Sanitizing***. So gelangen unveränderte Werte in unsere Datenbank. Bei der Ausgabe außerhalb von Formularen können wir zukünftig entscheiden, ob wir `htmlspecialchars` mit/ohne `strip_tags` oder den HTML-Purifier verwenden wollen.

Übung 53:

1. Passen Sie alle Formulare des Newstickers so an, dass die `e`-Funktion mit deaktiviertem ***Sanitizing*** verwendet wird.
2. Testen Sie das Kennwort `0123abc<DEF>` erneut.

Für Benutzereingaben, auf die man bei der Ausgabe ein `strip_tags` anwendet, kann gegebenenfalls eine Validierung zur Usability-Verbesserung sinnvoll sein, die die Zeichen `<` und `>` verbietet. Im Rahmen einer solchen Validierung könnte man beispielsweise den Original-Wert und den mit `strip_tags` bereinigten Wert vergleichen, bei einem Unterschied enthält der Original-Wert nicht erlaubte Zeichen. Durch eine solche Validierung vermeidet man, dass ein Benutzer sich später bei der Ausgabe über fehlende Zeichen wundert.

14.8 Ein paar grundsätzliche Hinweise

In den bisherigen Kapiteln dieser Lektion bin ich relativ ausführlich auf einzelne Themen eingegangen. Im Folgenden möchte ich kurz ein paar ergänzende Themenbereiche anreißen, damit Sie auch diese auf dem Radar haben.

14.8.1 Eval is Evil

Das Sprachkonstrukt eval() wertet eine Zeichenkette als PHP-Code aus.

Beispiel

```php
eval('echo "Hallo Welt!";');
```

eval() - Beispiel 1

Codebeispiel 173 *eval() - Beispiel 1*

Der Beispiel-Code würde tatsächlich den `echo`-Befehl ausführen, obwohl er nur als String vorliegt. Eigentlich ganz einfach, oder? Ja und nein. Solange es sich bei dem String nicht um eine Benutzereingabe handelt, haben Sie auch kein wirkliches Sicherheitsproblem. Falsch angewendet, ergeben sich für einen Angreifer jedoch einige interessante Möglichkeiten.

Beispiel

```php
$method = $_GET['method'];
eval('Foo::' . $method . '();');
```

eval() - Beispiel 2

Codebeispiel 174 *eval() - Beispiel 2*

In diesem Beispiel verwenden wir direkt eine Benutzereingabe aus `$_GET`, um eine statische Methode der Klasse `Foo` aufzurufen. Den Code habe ich so bei meinen Recherchen gefunden und er soll tatsächlich in der freien Wildbahn anzutreffen gewesen sein. Dies kann nun dadurch ausgenutzt werden, dass nicht nur der Name der Methode (z. B. `bar`) übergeben wird, sondern ähnlich wie bei einer SQL-Injection zusätzlicher Code angehängt wird.

Beispiel

```php
<?php

class Foo
{
    public static function bar()
    {
        // Der Inhalt ist nicht relevant
    }
}

$method = $_GET['method'];
eval('Foo::' . $method . '();');

```

```
14 ?>
15 <html>
16 <body>
17     <?php echo urldecode($method); ?><br/>
18     <?php var_dump($_SESSION); ?>
19 </body>
20 </html>
```

eval.php

Codebeispiel 175 *eval.php*

Versuchen Sie nun einmal, die Datei *eval.php* mit folgender Angabe aufzurufen `eval.php?method=bar%28%29%3B%24_SESSION%5B%22user_id%22%5D%3D1%3B%2F%2F`. Wenn der Aufruf korrekt war, so sollten Sie nun erkennen können, was die Attacke bewirkt hat. In diesem Fall wurde nämlich nicht nur die Methode `Foo#bar()` aufgerufen, sondern auch der User mit der ID 1 eingeloggt.

Aus gutem Grund hat *Rasmus Lerdorf* (der Erfinder von PHP) einmal geschrieben: »If eval() is the answer, you're almost certainly asking the wrong question« (dt. »Wenn eval() die Antwort ist, stellst Du sehr wahrscheinlich die falsche Frage«). Aber ist `eval()` wirklich böse? Persönlich stimme ich hierbei mit *Dan Horrigan* überein, der in einem Blog-Beitrag 2012 schrieb, dass man sich vor einer Verwendung fragen sollte: »Will this code run in production with user input?« Wenn es sich um ein produktives Umfeld mit Benutzereingaben handelt, sollten Sie `eval()` auf keinen Fall verwenden. Außer natürlich man weiß **wirklich** was man tut und ist **wirklich** sicher, dass der auszuführende Code absolut sicher ist.

Die Nutzung von `eval()` zum Aufruf einer variablen Funktion bzw. Methode wäre in den Beispielen dieses Kapitels übrigens nicht nötig gewesen, da wir schon einmal ganz zu Anfang bei den magischen Methoden (siehe »Band 1: Grundlagen der OOP«) eine bessere Alternative kennengelernt haben. Dort haben wir zum Aufruf der Setter die sogenannten ***Variablenfunktionen*** verwendet, d.h. wir können einfach `Foo::$method();` schreiben und somit in diesem Fall komplett auf `eval()` verzichten.

Sofern man `eval()` nicht benötigt, könnte man in Versuchung geraten, eine komplette Deaktivierung mit der *php.ini*-Direktive disable_functions

vorzunehmen. Da es sich bei `eval()` jedoch um ein ***Sprachkonstrukt*** und **nicht** um eine Funktion handelt, ist dies (derzeit) nicht möglich.

14.8.2 Datenminimierung

Schafft es trotz aller bisherigen Maßnahmen ein Angreifer, sich Zugriff auf die Datenbank einer Anwendung zu verschaffen, dann muss sich zunächst der Betreiber und dann auch der Entwickler für den Verlust der gespeicherten Daten verantworten. Aber auch sonst sollten Sie die Daten Ihrer Benutzer nicht einfach offen herumliegen lassen, jeder E-Mail-Harvester und Social Bot würde sich hierüber nämlich tierisch freuen. Beachten Sie bei der Umsetzung einer Web-Anwendung deswegen unbedingt ein paar einfache Regeln:

1. Speichern Sie wirklich nur die Daten, die für den Betrieb Ihrer Anwendung zwingend nötig sind.
2. Wenn Besucher sich Profile anderer Benutzer ansehen dürfen, so sollte dies vorzugsweise nur eingeloggten Benutzern möglich sein.
3. Lassen Sie jeden Benutzer über sogenannte Privatsphären-Einstellungen selbst entscheiden, welche Informationen (beispielsweise im Benutzerprofil) öffentlich zugänglich sind. Verwenden Sie hierbei sinnvolle und somit relativ restriktive Standardeinstellungen.
4. Ist die Anzeige einer sensitiven Information unvermeidlich, so sollte möglichst nur ein Teil dieser Information angezeigt werden. Amazon zeigt in der abschließenden Übersicht eines Bestellvorgangs beispielsweise nur die letzten zwei Stellen einer Kontonummer an. Dies entspricht weniger als 30% der Gesamtinformation und ist ein guter Richtwert.

14.8.3 HTTPS ist kein Allheilmittel

Secure Sockets Layer (SSL) ist ein hybrides Verschlüsselungsprotokoll zur Datenübertragung im Internet. Gleichzeitig kann ein Benutzer anhand eines sogenannten Zertifikats die Identität einer Website verifizieren. Ab Version 3.0 wurde das SSL-Protokoll unter dem Namen Transport Layer Security weiterentwickelt. TLS 1.0 meldet sich übrigens im Header noch als Version SSL

3.1.

TLS-Verschlüsselung wird heute vor allem bei ***Hypertext Transfer Protocol Secure*** eingesetzt. Bei HTTPS handelt es sich um ein Kommunikationsprotokoll zur abhörsicheren Datenübertragung zwischen Webserver und Browser. Ohne eine solche Verschlüsselung sind für jeden Angreifer, der Zugriff auf den Datenverkehr hat, alle übertragenen Daten im Klartext lesbar. Sobald sensitive Daten zwischen Browser und Webserver übertragen werden (beispielsweise Zahlungsinformationen) sollte immer HTTPS eingesetzt werden. Wenn es um sensitive Daten aus den Bereichen Finanzen und Gesundheit geht, so wird die Verwendung von HTTPS von den Benutzern grundsätzlich erwartet und das Fehlen des »intakten Schlosses« selten akzeptiert.

HTTPS ist jedoch kein Allheilmittel, da die benutzte Version von SSL/TLS von der Konfiguration des Webservers und dem verwendeten Browser bzw. Betriebssystem abhängig ist. Je älter diese Version ist, desto größer ist die Wahrscheinlichkeit, dass es inzwischen bekannte Angriffsmöglichkeiten gibt. Im Oktober 2014 wurde beispielsweise ein Angriff auf SSL 3.0 namens *Poodle* vorgestellt. Die entsprechende Sicherheitslücke war so gravierend, dass alle modernen Browser den Support von SSL 3.0 eingestellt haben und nur noch TLS unterstützen.

- https://www.heise.de/security/meldung/Poodle-Experten-warnen-vor-Angriff-auf-Internet-Verschluesselung-2424122.html
- https://www.globalsign.com/de-de/blog/internet-explorer-mozilla-firefox-und-google-chrome-deaktivieren-standardmassig-ssl-30/
- https://en.wikipedia.org/wiki/Template:TLS/SSL_support_history_of_web_b

Selbst wenn Sie eine noch nicht angreifbare TLS-Version nutzen sollten, schützt dies jedoch lediglich die Übertragung der Daten (SSL/TLS ist ein »auf-der-Leitung« Protokoll). Sind die Daten erstmal auf dem Webserver bzw. im Browser angekommen, so können sie dort dennoch eine schädliche Wirkung entfalten.

14.9 Letzte Worte

In den komplexen »Öko-Systemen«, in welchen sich Web-Anwendungen heutzutage bewegen, reicht eine alleinige Absicherung der Anwendung nicht aus. Auch die restlichen Software-Komponenten, die Hardware (Server- und Netzwerkkomponenten) und die verwendeten Räumlichkeiten sollten neben den eigentlichen Benutzern Bestandteil eines **umfassenden Sicherheitskonzeptes** sein. Schließlich könnte auch ein Angreifer, der über einen dieser Wege Zugriff erlangt, für Sie und Ihre Anwendung gefährlich werden.

Bedenken Sie zudem, dass Sie im Falle eines Falles immer ein aktuelles ***Backup*** Ihrer Anwendung zur Hand haben sollten. Dies betrifft sowohl die Dateien der Anwendung (inklusive möglicherweise vorhandenen Benutzer-Uploads) als auch die Inhalte der Datenbank. Es nützt übrigens nicht viel, wenn eine solche Datensicherung auf dem gleichen Gerät wie die eigentliche Anwendung gespeichert wird und somit beides von einem Hardwaredefekt betroffen wäre. Am besten ist es sogar, wenn Ihre Sicherungen an einem komplett anderen Ort aufbewahrt werden, damit ein Feuer nur eines von beidem vernichten kann.

Sie sollten auch immer die Neuigkeiten in der PHP-Welt im Blick behalten. So sind Sie frühzeitig informiert, wenn es gravierende Änderungen am Öko-System Ihrer Anwendung geben sollte, und können ohne Zeitdruck die nötigen Anpassungen planen und umsetzen. Denn nichts ist schlimmer, als wenn eine Anwendung nach dem Update einer Komponente für längere Zeit ausfällt. Persönlich informiere ich mich beispielsweise möglichst täglich.

- http://phpdeveloper.org/archive
- https://entwickler.de/online/php

Nehmen Sie sich abschließend einen Satz aus dem fliegenden Klassenzimmer von *Erich Kästner* zu Herzen: »An jedem Unfug, der passiert, sind nicht nur die schuld, die ihn begehen, sondern auch die, die ihn nicht verhindern.« Versuchen Sie also immer das Möglichste, um die Integrität Ihrer Anwendung nicht zu gefährden.

14.10 Testen Sie Ihr Wissen

1. Welche Daten eines Benutzers sind sicher?
2. Ist eine absolute Risiko-Eliminierung bei einer Web-Anwendung möglich?
3. Welche fünf Tugenden sollte sich ein sicherheitsbewusster Entwickler angewöhnen?
4. Unter welcher Prämisse sollte man heutzutage Anwendungen entwickeln?
5. Wieso ist ein Whitelist-Ansatz einer Blacklist meist überlegen?
6. Was ist die primäre Maßnahme zum Schutz einer Anwendung gegen SQL-Injections?
7. Was ist der Unterschied zwischen persistentem und reflektiertem Cross-Site-Scripting?
8. Welche Möglichkeiten hat ein Angreifer, um an das Kennwort eines Benutzers zu gelangen? Gehen Sie in diesem Fall von einer Anwendung ohne gravierende Sicherheitslücken aus.
9. Was ist der wichtigste Unterschied zwischen Verschlüsselung und Hashing?
10. Reicht HTTPS als alleinige Verteidigungslinie für eine Web-Anwendung aus?

14.11 Aufgaben zur Selbstkontrolle

Beispiel

```php
<?php

function logIn($id)
{
    $_SESSION['user_id'] = $id;
}

function isLoggedIn()
{
    return (isset($_SESSION['user_id']) && !empty($_SESSION[
```

```
11 }
12
13 function logOut()
14 {
15     unset($_SESSION['user_id']);
16 }
```

inc/functions.inc.php

Codebeispiel 176 *inc/functions.inc.php*

Übung 54:

Implementieren Sie den letzten Stand des Templates *loginAction.tpl.php* und die drei Funktionen `logIn()`, `isLoggedIn()` und `logOut()` in Ihrem Projekt. Außerdem benötigen wir im Template *navi.tpl.php* unter `Register` einen neuen Link `Login`.

Übung 55:

Setzen Sie nun die Methode `UserController#loginAction()` um. Wenn Formulardaten vorhanden sind, soll sie ein Objekt der `User`-Entity anhand der eingegebenen E-Mail-Adresse ermitteln. Wird ein Objekt gefunden und stimmt das Password mit der Formulareingabe überein, so soll die Funktion `logIn()` aufgerufen werden. Als Parameter müssen Sie hierbei die ID des Objekts verwenden. Erzeugen Sie außerdem eine Flash-Notice, die den erfolgreichen Login bestätigt. Machen Sie dann eine Header-Umleitung zur Startseite.

Falls Formulardaten vorhanden sind, diese aber nicht korrekt sind, so befüllen Sie die Variable `$errors` manuell mit einem Array. Dieses Array soll als einzigen Wert den String `'Fehlerhafte Logindaten!'` haben. Da wir im Fehlerfall keine Header-Umleitung machen, benutzen wir zur Anzeige dieser Meldung auch keine Flash-Notice.

Übung 56:

Passen Sie nun die *navi.tpl.php* erneut an. Die `Register`- und `Login`-Links

sollen nur noch angezeigt werden, wenn der User noch nicht eingeloggt ist. Ist er eingeloggt, so sollen stattdessen die Links zum Erstellen von Tags bzw. Articles und ein Logout-Link angezeigt werden. Benutzen Sie als Bedingung die Funktion `isLoggedIn()`.

Übung 57:

Benutzen Sie diese Funktion nun auch, um die Links zum Bearbeiten und Löschen von `Article`-Datensätzen nur noch bei eingeloggten Benutzern anzuzeigen.

Übung 58:

Setzen Sie die Methode `UserController#logoutAction()` um. Diese soll den User mit der Funktion `logOut()` ausloggen, eine Flash-Notice anlegen und dann auf die Startseite des Newstickers weiterleiten. Ein Template benötigen Sie somit nicht.

14.12 Optionale Aufgaben

Übung 59:

Das Begleitmaterial enthält einen »finalen« Stand des Newstickers. Man könnte jedoch immer noch vieles an diesem Stand verbessern. So fehlt beispielsweise noch eine Limitierung der Anmeldeversuche, um eine weitere Verteidigungslinie gegen Brute-Force-Attacken zu haben. Außerdem wird das Attribut `$publishAt` zwar befüllt, aber noch nirgends wirklich verwendet.

Der »finale« Stand verwendet **Passwort-Hashes**, ein zusätzliches Attribut `$displayName` in der `User`-Entity (inkl. Validierung) und **beschränkt** in den Controller-Klassen den **Zugriff auf manche Aktionen**. Ein Verstecken der Links reicht alleine niemals aus, es muss auch ein Aufruf bei manuell eingegebenen URLs verhindert werden. Ich habe mich in der Musterlösung

ausnahmsweise für eine Blacklist entschieden, da aktuell nur ein Schutz der Aktionen add, edit und delete nötig ist. Außerdem wird nun endlich der korrekte `User` als `Article`-Ersteller in der Datenbank hinterlegt und alle Formulare sind mit einem CSRF-Token geschützt. Installieren Sie diesen Stand, welcher die neue Datenbank `newsticker_final` verwenden soll, auf Ihrem Entwicklungssystem. Denken Sie hierbei daran, die Datenbanktabellen durch einen Aufruf der *setup.php* zu erstellen und diese mit der *reset.php* zu befüllen. Welche Datei verhindert nochmal die Ausführung dieser beiden Controller im Browser?

Vollziehen Sie die Änderungen bzw. Inhalte der nachfolgenden Dateien nach und schauen Sie sich auch ruhig mal den Inhalt der Datenbank genauer an:

- *config/default-config.php* (Verbesserung der Session-Sicherheit)
- *inc/functions.inc.php* (u.a. Verbesserung der Session-Sicherheit)
- *src/Controllers/AbstractSecurity.php* (**neue** Elternklasse)
- *src/Controllers/IndexController.php* und *src/Controllers/TagController.php* (u.a. Zugriffsbeschränkungen durch Verwendung der neuen Elternklasse)
- *src/Controllers/UserController.php* (u.a. Änderungen in der `login`-Aktion)
- *src/Entities/User.php*
- *src/Repositories/UserRepository.php* (flexiblere Duplikat-Methode)
- *src/Validators/AbstractSecurity.php* (**neue** Elternklasse)
- *src/Validators/ArticleValidator.php*
- *src/Validators/TagValidator.php*
- *src/Validators/TokenValidator.php* (u.a. für CSRF-Token-Prüfung in der `login`-Aktion)
- *src/Validators/UserValidator.php* (leeres Kennwort beim Editieren erlaubt)
- *templates/IndexController/deleteAction.tpl.php*
- *templates/IndexController/editAction.tpl.php*

- *templates/TagController/editAction.tpl.php*
- *templates/UserController/editAction.tpl.php* (u.a. neues Eingabefeld)
- *composer.json* (**neues** Paket zur Erzeugung der CSRF-Tokens)
- *reset.php* (Befüllung neues Attribut).

15 Anhang: Weiterführende Informationen

In dieser Lektion lernen Sie
• wo Sie interessante, weiterführende Quellen zum Thema PHP finden.

15.1Einführung

Nach dem langen Weg durch dieses Lernbuch muss ich Ihnen leider sagen, dass das noch lange nicht alles war, was es zum Thema PHP, OOP oder gar Sicherheit zu wissen gibt. Sonst wäre dieses Buch mehrere tausend Seiten dick und das Wort »alles« wäre immer noch gelogen. Deshalb habe ich Ihnen in der letzten Lektion einige interessante Webseiten und Bücher zusammengestellt, über die Sie weiterführende Informationen rund um PHP erhalten.

15.2 Weblinks

15.2.1 www.php.net

http://php.net/manual/de/

Die offizielle Seite der PHP-Entwickler bietet nicht nur eine Übersicht über alle PHP-Funktionen, sondern auch ein Handbuch in deutscher Sprache. Dieses Handbuch versteht sich selbst eher als Referenz denn als Lernheft. Folglich sind auch die Erklärungen und Beispiele effizient, aber eher knapp gehalten. Die Seite eignet sich sehr gut, um in Ihrem PHP-Wissen gezielt Lücken zu füllen. Als entspannte Abend-Lektüre würde ich sie weniger empfehlen.

15.2.2 www.phpdeveloper.org

http://www.phpdeveloper.org/

<?PHPDeveloper.org ist eine englischsprachige Kommunikationsplattform rund um PHP. Sie finden dort eine Übersicht von Neuigkeiten aus der PHP-Welt, Informationen über Konferenzen, Jobangebote, aktuelle Diskussionen und eine Übersicht der interessantesten Fachartikel der letzten Zeit.

15.2.3 devzone.zend.com

http://devzone.zend.com/

In der englischsprachigen Entwickler-Community von ***ZEND***, den Entwicklern von PHP, finden Sie interessante und oft sehr aktuelle, aber doch recht fortgeschrittene Artikel (engl. Tutorials) zum Thema PHP. Die Seite wird für Sie umso wichtiger werden, je mehr Sie über PHP wissen.

15.3 Buchtipps

Die Lektion zum Thema Sicherheit basiert auf den nachfolgenden Quellen. Die Lektion bietet jedoch aus Platzgründen lediglich einen Querschnitt der dortigen Themen.

Christopher Kunz, Stefan Esser, Peter Prochaska: PHP-Sicherheit. PHP/MySQL-Webanwendungen sicher programmieren. 2. aktualisierte und überarbeitete Auflage, dpunkt.verlag 2007 (ISBN-13 978-3898644501)

Tobias Wassermann: Sichere Webanwendungen mit PHP, mitp 2007 (ISBN-13 978-3826617546)

Chris Snyder, Thomas Myer, Michael Southwell: Pro PHP Security. From Application Security Principles to the Implementation of XSS Defenses. Second Edition, Apress 2010 (ISBN-13 978-1430233183)

> Ich kann Ihnen nur ans Herz legen: Verlassen Sie sich nicht nur auf meine Kurzübersicht, sondern lesen Sie die ausführlicheren Originale (auch wenn

sie oftmals etwas veraltet sind)!

Lösungen der Wissensfragen

Lektion 2: Objekt-relationales Mapping

1. **Wie viele Klassen repräsentieren normalerweise eine Datenbank-Tabelle, wenn Sie Active Record verwenden?**

   Nur eine.

2. **Wie viele Objekte erhalten Sie in PHP, wenn Sie mit Active Record 30 Datensätze aus der DB auslesen?**

   30 Objekte, eines für jeden Datensatz.

3. **Wie viele Klassen repräsentieren normalerweise eine Datenbank-Tabelle, wenn Sie Data Mapper verwenden?**

   Zwei Klassen, eine für die Datenhaltung und eine für die Datenverwaltung (Speichern, Löschen usw.).

4. **Wie viele Objekte erhalten Sie in PHP, wenn Sie mit dem Data Mapper 30 Datensätze aus der DB auslesen?**

   31! 30 Objekte der Datenklasse und ein Objekt des Mappers.

Lektion 3: Composer, Packagist & Co.

1. **Welche Konsolenkommandos von Composer haben Sie kennengelernt?**

   - `php composer.phar about`
   - `php composer.phar self-update`
   - `php composer.phar create-project`
   - `php composer.phar search`
   - `php composer.phar browse`

    - `php composer.phar install`

2. **Was erspart Ihnen eine `__autoload()`-Funktion?**

   Eine solche Funktion erspart hauptsächlich viel Schreibarbeit, indem sie einen Automatismus für die Einbindung von Klassen festlegt. Zudem erübrigt sich die vorherige Festlegung der benötigten Klassen, da der Automatismus genau in dem Moment greift, wo die Klasse beispielsweise im Rahmen einer Instanziierung benötigt wird.

Lektion 4: Doctrine-Entities

1. **Nennen Sie Beispiele für Namespaces jenseits der PHP-Programmierung.**

   z. B. Dateiverzeichnisse, Postleitzahlen und Telefonvorwahlen

2. **Wir verwenden Annotationen zur Festlegung des Datenbank-Schemas. Was ist der primäre Unterschied zu den drei anderen Varianten?**

   Annotationen erfolgen in Form von **Kommentaren** direkt **in den Klassen**. Die restlichen Varianten werden in separaten Dateien umgesetzt.

3. **Überlegen Sie, ob jedes Attribut Annotationen (z. B. `@ORM\Column`) benötigt.**

   Attribute, die **nicht** von Doctrine in der Datenbank gespeichert werden sollen, benötigen auch **keine** Annotationen. Solche Attribute sind allerdings vergleichsweise selten.

Lektion 5: Aufbau einer Datenbank-Verbindung

1. **Wie nennt man die Schnittstelle zu Doctrine, die man zum Auslesen, Speichern und Löschen von Datensätzen nutzt?**

   EntityManager

2. **Es gibt eine Namenskonvention für die Variable, in der das Objekt dieser Schnittstelle liegt. Wie lautet diese?**

`$em`

Lektion 6: PHP-Objekte mit Doctrine speichern

1. **Reicht es, die Methode `EntityManager#persist()` aufzurufen, um ein Objekt zu speichern?**

   Nein. Erst wenn man die Methode `EntityManager#flush()` aufruft, werden alle SQL-Anweisungen ausgeführt, die sich bis zu diesem Zeitpunkt angesammelt haben.

2. **Wie nennt man das Vorgehen, Aufgaben zu sammeln und am Stück abzuarbeiten?**

   Unit of Work

Lektion 7: Datenbankabfragen mit Doctrine

1. **Was repräsentiert bei Doctrine eine Tabelle mit allen enthaltenen Datensätzen und dient gleichzeitig dazu, Datensätze nach bestimmten Kriterien auszulesen?**

   Die Tabelle wird durch ihr EntityRepository repräsentiert, welches man mit der Methode `EntityManager#getRepository()` erhält.

2. **Man legt das EntityRepository selten in einer eigenen Variablen ab, sondern hängt einen weiteren Methodenaufruf an den ersten an. Wie nennt man diese Vorgehensweise?**

   Man nennt diese Vorgehensweise Chaining und kombiniert sie zur besseren Lesbarkeit oftmals mit manuellen Zeilenumbrüchen, d.h. jeder Methodenaufruf steht in einer eigenen Zeile.

Lektion 8: Komplexe Abfragen mit Doctrine

1. **Wie nennt sich das bei Doctrine verwendete Äquivalent zu SQL und was ist der wichtigste Unterschied zu SQL?**

Bei Doctrine verwendet man für Datenbankabfragen DQL (Doctrine Query Language). Der Hauptunterschied besteht darin, dass man mit DQL nicht Tabellen befragt, sondern die Entities.

2. **Was verwendet man als Rückgabewert der Methoden, um in einer Klasse ein einfaches Fluent Interface umzusetzen?**

   `$this`

Lektion 9: Die Webmasters Doctrine Extensions

1. **Welches Konfigurations-Array ist durch das überarbeitete Bootstrapping nun optional?**

   `$applicationOptions`

2. **Welches Paket haben wir neben den *Webmasters Doctrine Extensions* in dieser Lektion noch installiert?**

   Gedmo Doctrine Extensions

Lektion 10: DateTime

1. **Welche Alternative(n) zu der Unmenge von Datums- und Zeitfunktionen haben Sie in dieser Lektion kennengelernt?**

   Die Klasse `DateTime` bzw. `Webmasters\Doctrine\ORM\Util\DateTime`

2. **Womit kann man automatisiert ein Erstellungs- oder Aktualisierungsdatum in einem Attribut pflegen?**

   Die automatisierte Datumspflege ist mit der Annotation `@Gedmo\Timestampable` möglich, die jedoch **nicht** Doctrine-Standard ist und erst durch die Nutzung des Pakets `gedmo/doctrine-extensions` zur Verfügung steht.

Lektion 11: Datenbank-Beziehungen mit Doctrine

1. **Wie definiert man Beziehungen zwischen Doctrine-Entities?**

Man benötigt in beiden Entities ein neues Attribut und versieht dieses mit Annotationen.

2. **Genauer gefragt, was nutzt man, um eine 1:n-Beziehung abzubilden?**

   Auf der 1(One)-Seite der Beziehung nutzt man die Annotation `@ORM\OneToMany` (Attributs-Bezeichner im Plural) und auf der n-Seite benötigt man ein `@ORM\ManyToOne` (Attributs-Bezeichner im Singular).

3. **Und was nutzt man, um eine n:m-Beziehung abzubilden?**

   Auf beiden Seiten nutzt man eine `@ORM\ManyToMany`-Annotation (Attributs-Bezeichner im Plural). Diese unterscheiden sich allerdings beim Bezeichner eines Parameters (`mappedBy` auf der Gegenseite bzw. `inversedBy` auf der Eigentümerseite).

4. **Welche Methoden der Klasse `ArrayCollection` haben Sie kennengelernt?**

   Sieben Methoden der Klasse `ArrayCollection` sollten Sie kennen, nämlich `clear`, `add`, `contains`, `removeElement`, `toArray`, `isEmpty` und `count`.

5. **Was ist Lazy Loading?**

   Als Lazy Loading bezeichnet man das Nachladen von Daten bei Bedarf. Bei Doctrine wird dies durch die Nutzung sogenannter Proxy-Klassen ermöglicht.

Lektion 12: Controller-Klassen im Überblick

1. **Wofür steht das Akronym CRUD?**

   Das Acronym steht für die Datenbankoperationen Create, Read, Update und Delete.

2. **Wofür steht das Akronym BREAD?**

   Das Acronym steht für die Datenbankoperationen Browse, Read, Edit, Add und Delete.

3. **Weswegen beschreibt BREAD die Benutzeroberfläche einer MVC-Anwendung besser als CRUD?**

BREAD berücksichtigt den Code-Unterschied bei der Anzeige eines bzw. mehrerer Datensätze und trennt deswegen Read und Browse in zwei verschiedene Operationen.

4. **Wodurch unterscheiden sich die Operationen Read, Edit und Delete von den (eher allgemeinen) Operationen Browse und Add?**

   Read, Edit und Delete benötigen zur Ausführung die ID eines Datensatzes.

Lektion 13: Fortgeschrittene Techniken

1. **In welcher Datei kann klassenbasierter Validierungscode für unsere `User`-Entity abgelegt werden und wo wird diese Datei gespeichert?**

   Entsprechender Code sollte in *src/Validators/UserValidator.php* abgelegt werden, sofern man die Webmasters Doctrine Extensions verwenden möchte.

2. **Wozu werden eigene Repository-Klassen benutzt?**

   In benutzerdefinierten Repository-Klassen können (umfangreichere) Datenbankabfragen ausgelagert werden, wie sie beispielsweise bei der Nutzung des QueryBuilders und JOINs auftreten. Bei einem gut gewählten Methodennamen verbessert sich so auch die Lesbarkeit des Controller-Codes und es wird eine Wiederverwendbarkeit der entsprechenden Datenbankabfrage ermöglicht.

Lektion 14: Eine Einführung in das Thema Sicherheit

1. **Welche Daten eines Benutzers sind sicher?**

   Daten, die gar nicht erst erhoben und gespeichert werden.

2. **Ist eine absolute Risiko-Eliminierung bei einer Web-Anwendung möglich?**

   Theoretisch ja, praktisch nein. Man spricht deshalb lediglich von einem Risiko-Management.

3. **Welche fünf Tugenden sollte sich ein sicherheitsbewusster Entwickler angewöhnen?**

1. Nichts ist zu 100 Prozent sicher.
2. Traue niemals den Benutzern.
3. Benutze immer mehrere Verteidigungslinien.
4. Wartbaren Code kann man besser absichern.
5. Vier Augen sehen mehr als zwei.

4. **Unter welcher Prämisse sollte man heutzutage Anwendungen entwickeln?**

   »Der Feind kennt das System.«

5. **Wieso ist ein Whitelist-Ansatz einer Blacklist meist überlegen?**

   Bei einer Blacklist benötigt man eine genaue Kenntnis der nicht erlaubten Angaben, bei einer Whitelist hingegen schaltet man selektiv nach und nach die erlaubten (und getesteten) Werte frei.

6. **Was ist die primäre Maßnahme zum Schutz einer Anwendung gegen SQL-Injections?**

   Prepared Statements mit (benannten) Platzhaltern für alle Benutzereingaben sind der wichtigste Schutz. Ihre Nutzung ist jedoch nur möglich, sofern die Eingabe als Wert (und z. B. nicht als Spalte bzw. Attribut) im SQL-/DQL-Statement benötigt wird.

7. **Was ist der Unterschied zwischen persistentem und reflektiertem Cross-Site-Scripting?**

   Beim persistenten XSS erfolgt der Angriff auf den Benutzer bei einem normalen Aufruf der Website. Der Schadcode ist dauerhaft in die Website integriert. Beim reflektierten XSS muss die Website hingegen auf eine spezielle Art und Weise aufgerufen werden (meist über einen präparierten Link). Der Schadcode gelangt erst über diesen Aufruf in die Ausgabe der Website.

8. **Welche Möglichkeiten hat ein Angreifer, um an das Kennwort eines Benutzers zu gelangen? Gehen Sie in diesem Fall von einer Anwendung ohne gravierende Sicherheitslücken aus.**

   Wörterbuchangriff, Brute-Force-Methode oder einfach eine Tafel Schokolade ;-)

9. **Was ist der wichtigste Unterschied zwischen Verschlüsselung und Hashing?**

   Bei einem Einweg-Hash-Algorithmus ist im Gegensatz zu einer Verschlüsselung der Vorgang nicht umkehrbar.

10. **Reicht HTTPS als alleinige Verteidigungslinie für eine Web-Anwendung aus?**

    Nein, HTTPS schützt lediglich die Übertragung der Daten. Diese Daten können aber immer noch Schadcode enthalten.

Über den Autor

Jan Teriete ist seit 1997 in der IT tätig. Vom EDV-Support erfolgte relativ schnell der erste Schritt in die Weiten des World Wide Web. In den Jahren 2004/2005 absolvierte er eine Vollzeit-Ausbildung zum Webmaster, womit auch seine Liebe zu PHP begann.

Seit 2005 ist Jan Teriete als Freelancer unterwegs und verdient seinen Lebensunterhalt als Web-Entwickler, vorwiegend mit CMS-Projekten auf Basis von Drupal und WordPress. Nebenbei ist er auch bei der Webmasters Akademie in Nürnberg als freiberuflicher Tutor, Dozent und Autor für den Fachbereich PHP/OOP und Doctrine 2 tätig.

Außerdem ist er bei Drupal- und WordPress-Meetups im Raum Nürnberg anzutreffen, wo er regelmäßig Vorträge zu CMS-Themen hält.

Jetzt Buch registrieren und Begleitmaterial herunterladen!

Registrieren Sie Ihr Buch auf webmasters-press.de und laden Sie sich die Übungsdateien und Lösungen herunter:

www.webmasters-press.de/register
Registrierungscode: 12e80c5577b2

Table of Contents