

FACHHOCHSCHULE FLENSBURG

Fachbereich Angewandte Informatik

BACHELORTHESIS

im Studiengang Medieninformatik

Thema: Integration der Datenbank-Abstraktionsschicht Doctrine2
in das Content-Management-System TYPO3

eingereicht von: Stefan Kowalke <stefan.kowalke@stud.fh-flensburg.de>

Matrikelnummer: 485366

Abgabedatum: 23. Mai 2014

Erstprüfer: Prof. Dr. Hans-Werner Lang

Zweitprüfer: Dipl. VK Tobias Hiep

Dieses Dokument wurde am 23. Mai 2014 mit L^AT_EX 2_ε gesetzt.

Schrift: 12pt-TODO
Typographie: 2012/07/29 v3.11b KOMA-Script
System: LuaT_EX-0.76.0 auf OSX 10.8
Editor: Mou 0.8.5 beta und VIM 7.4

Dieses Werk steht unter der **Creative Commons Namensnennung – Weitergabe unter gleichen Bedingungen 4.0 International Lizenz**.

Es kann unter <https://github.com/Konafets/thesis> heruntergeladen werden.



Der für diese Thesis entstandene Prototyp steht unter der GNU General Public License version 2 oder neuer.

The Joy of Programming with Bob Ross von Abstruse Goose zu finden unter <http://abstrusegoose.com/467> steht unter der Lizenz **Attribution-NonCommercial 3.0 United States (CC BY-NC 3.0 US)** <http://creativecommons.org/licenses/by-nc/3.0/us/>

Exploits of a mom von xkcd zu finden unter <http://xkcd.com/327/> steht unter der Lizenz **Attribution-NonCommercial 2.5 Generic (CC BY-NC 2.5)** <http://creativecommons.org/licenses/by-nc/2.5/>

ABSTRACT

Webanwendungen werden häufig um ein Datenbankmanagementsystem herum entworfen. In der Vergangenheit war dies oft MySQL. Um eine Webanwendung aus der Abhängigkeit zu einem spezifischen Datenbankmanagementsystem zu lösen, kann die, von PHP mitgelieferte, Datenbankabstraktionsschicht PDO genutzt werden. Einen Schritt weiter geht Doctrine DBAL, welches – auf PDO aufbauend – eine einheitliche Schnittstelle zu weiteren Datenbankmanagementsystemen bereitstellt. Doctrine DBAL ist zudem die Grundlage für Doctrine ORM - ein Framework zur Objekt-relationalen Abbildung von Objekten auf eine Datenbank, das in TYPO3 Flow und TYPO3 Neos eingesetzt wird. Diese Arbeit demonstriert wie durch die Integration von Doctrine DBAL in das Content-Management-System TYPO3 CMS die Abhängigkeit zu MySQL entfernt werden kann.

“War is peace.
Freedom is slavery.
Ignorance is strength.”

– George Orwell, 1984

INHALTSVERZEICHNIS

Abstract	iii
1 Einleitung	1
2 Grundlagen	4
2.1 TYPO3 CMS	4
2.1.1 Geschichte	4
2.1.2 Definition	5
2.1.3 Architektur und Aufbau von TYPO3 CMS	5
2.1.4 Extensions	7
2.2 Doctrine	8
2.2.1 ORM	9
2.2.2 DBAL	10
2.2.3 Unterstützte Datenbank-Plattformen	13
2.2.4 Konzepte und Architektur von Doctrine	13
2.3 Arbeitsweise	25
2.3.1 Formatierung des Quellcodes	25
2.3.2 Unit Testing	26
2.3.3 Versionsverwaltung	27
2.3.4 Travis-CI	27
2.3.5 IDE	28
2.4 Aktuelle Situation	29
2.4.1 Prepared Statements	31
2.4.2 Datenbankschema	32
3 Prototypischer Nachweis der Herstellbarkeit	36
3.1 Konzeption des Prototypen	36
3.2 Vorbereitung	37
3.2.1 Installation von TYPO3 CMS	37
3.2.2 Schritt 1 - Systemcheck	38
3.2.3 Schritt 2 - Eingabe der Datenbankdaten	38
3.2.4 Schritt 3 - Auswahl der Datenbank	38
3.2.5 Schritt 4 - Einrichten eines TYPO3 Administrators	38
3.2.6 Schritt 5 - Abschluß der Installation	38
3.2.7 Implementation von Unit Tests der alten Datenbank API	40
3.3 Erstellung des Prototypen	41
3.4 Integration des Prototypen in das Install Tool	46
3.4.1 Doctrines Schemarepräsentation	48
3.4.2 Installation des Prototypen	50
3.5 Umstellen der Query-Methoden auf Doctrine DBAL und Prepared Statements	52
Glossar	60
Abkürzungsverzeichnis	61
Eidesstattliche Erklärung	65

EINLEITUNG

Als sich auf den Developer Days 2006 das Entwicklerteam für einen Nachfolger der eben erst erschienenen TYPO3 Version 4.0 formierte (vgl. [TYP08]), war wohl keinem der dort Anwesenden klar wohin die Reise gehen würde – ging man anfänglich noch von einem Refactoring¹ der schon vorhandenen Codebasis aus.

In der Konzeptionsphase kristallisierte sich immer mehr heraus, dass es damit nicht getan sein würde. Der Nachfolger mit dem Arbeitstitel “Phoenix” sollte nicht nur den zukünftigen Anforderungen des Web standhalten, sondern die Position der Version 4.0 weiter ausbauen. Das Entwicklerteam um Chefentwickler Robert Lemke entschloss sich die Version 5.0 des Systems komplett neu zu schreiben [Quelle anfügen] und merkte dabei, dass Entwickler bei der Programmierung von Webanwendungen immer wieder mit den gleichen Problemen wie Routing, die Erstellung und Validierung von Formularen, Login von Benutzern oder dem Aufbau einer Verbindung zur Datenbank konfrontiert werden.

Die Idee eines – von dem Content-Management-System – unabhängigen PHP Frameworks war geboren und wurde zunächst auf den Namen FLOW3 getauft. Dieses Framework sollte die spätere Basis für TYPO3 5.0 bilden und all die oben beispielhaft angeführten wiederkehrenden Aufgaben übernehmen. Die Version 5.0 von TYPO3 sollte lediglich eins von vielen Packages darstellen mit denen FLOW3 erweitert werden kann. Vielmehr wurde es als eigenständiges “Webapplication Framework” konzipiert und umgesetzt, so dass es auch ohne ein Content Management-System (CMS) betrieben werden kann und auch wird. [Quelle zu Rossmann einfügen].

Schon in einer recht frühen Entwicklungsphase hat man sich dem Thema Persistenz gewidmet, die zunächst noch als “Content Repository for Java Technology API (JCR)” in PHP implementiert, jedoch später wegen zu vieler Probleme bei der Portierung der Java Spezifikation JSR-170 nach PHP durch eine eigene Persistenzschicht ersetzt wurde (vgl. [Dam10]). Im weiteren Verlauf der Entwicklung kam man von dieser Idee wieder ab, da die eigene Persistenzschicht nicht performant genug war und andere Projekte wie Doctrine oder Propel schon fertige Lösungen anboten (vgl. [DEM14]). Schließlich entschied man sich für die Integration von Doctrine als Persistenzschicht, da der Hauptentwickler von Doctrine, Benjamin Eberlei, seine Hilfe anbot.

Für die Anwender stellt sich bei einem Versionssprung stets die Frage, ob eine Migration von der alten zur neuen Version möglich ist und mit wieviel Aufwand dies verbunden sein würde. Diesen Bedenken folgend trafen sich die Kernentwickler beider Teams 2008 in Berlin, um die Routemaps beider Projekte in Einklang zu bringen. Als ein Ergebnis dieses Treffens wurde das “Berlin Manifesto”(vgl. [TYP08]) bekanntgegeben, welches mit klaren Worten feststellt²:

¹ Strukturverbesserung des Quellcodes bei Beibehaltung der Funktionalität

² Mittlerweile wird TYPO3 Neos innerhalb der Community nicht mehr als der Nachfolger von TYPO3 CMS angesehen. Es stellt lediglich – wie TYPO3 Flow – ein weiteres Produkt innerhalb der TYPO3 Familie dar. [Quellen angebe]



- TYPO3 v4 continues to be actively developed
- v4 development will continue after the the release of v5
- Future releases of v4 will see its features converge with those in TYPO3 v5
- TYPO3 v5 will be the successor to TYPO3 v4
- Migration of content from TYPO3 v4 to TYPO3 v5 will be easily possible
- TYPO3 v5 will introduce many new concepts and ideas. Learning never stops and we'll help with adequate resources to ensure a smooth transition

Die TYPO3 Kernentwickler

An der Umsetzung wurde sofort nach dem Treffen begonnen, indem Teile des FLOW3 Frameworks nach TYPO3 Version 4.0 zurück portiert und unter dem Namen *Extbase* als Extension veröffentlicht wurden. Es erfüllt zu gleichen Teilen die Punkte 3 und 6 des Manifests, da es die neuen Konzepte aus FLOW3 der Version 4.0 zur Verfügung stellt und somit gleichzeitig diese Version näher an die Technologie des Frameworks heranführt.

Die Aufgabe von Extbase besteht darin ein Application Programming Interface (API) bereitzustellen, mit denen Entwickler von Extensions auf die internen Ressourcen und Funktionen von TYPO3 CMS zugreifen und das System somit nach eigenen Wünschen und Anforderungen erweitern können, ohne den Code des CMS selbst verändern zu müssen. Es ist als vollständiger Ersatz der bis dahin angebotenen PI-BASE API [LINK ZU PI BASE] konzipiert worden, wobei es aktuell noch möglich ist sich für einen der beiden Ansätze zu entscheiden.

Extbase führt per Definition einige – bis dahin in TYPO3 v4 unbekannte – Programmierparadigmen ein. Als größter Unterschied zu dem PI-Based Ansatz ist hier sicherlich das Model-View-Controller (MVC) Pattern zu nennen. Dabei werden die Daten im Model vorgehalten, der View gibt die Daten aus und der Controller steuert die Ausgabe der Daten. Das Model ist unabhängig von der View, was bedeutet, dass die gleichen Daten auf verschiedene Weise ausgegeben werden können. Man denke hier an Meßdaten, die zum einen als Tabelle über einer Listview dargestellt werden können oder als Diagramme mit einer entsprechenden View.

Das Model – eine herkömmliche PHP Klasse – wird dabei von Extbase automatisch auf die Datenbank abgebildet, so dass ein Objekt eine Zeile darstellt und dessen Eigenschaften als Spalten der Tabelle interpretiert werden. Diese Technik wird als Objektrelationale Abbildung (engl. Object-relational mapping (ORM)) genannt. Das zum Einsatz kommende ORM ist Bestandteil der oben erwähnten selbstgeschriebenen Persistenzschicht von FLOW3, da Extbase zu der Zeit rückportiert wurde, als diese bei FLOW3 im Einsatz war.

Obwohl Extbase beständig weiterentwickelt wird und es der Wunsch der Community ist, die in darin verwendete Persistenzschicht gegen Doctrine 2 auszutauschen, was sich in Form von Posts auf der Mailingliste (vgl. [TYP13]) oder in Prototypen ausdrückt (vgl. [Mar12] und [Ebe12]), ist dies bis heute noch nicht realisiert worden. Der Chefentwickler von Doctrine, Benjamin Eberlei, hat gegenüber dem Autor in einer persönlichen

Korrespondenz die unterschiedlichen Ansätze beider Projekte wie folgt zum Ausdruck gebracht:

“ (...) Doctrine nutzt das Collection interface, Extbase SplObjectStorage. Doctrine Associationen funktionieren semantisch anders als in Extbase, z.B. Inverse/Owning Side Requirements. Typo3 hat die Enabled/Deleted flags an m_n tabellen, sowie das start_date Konzept. Das gibts in Doctrine ORM alles evtl nur über Filter API, aber vermutlich nicht vollständig abbildbar. Das betrifft aber alles nur das ORM, das Doctrine Database Abstraction Layer (DBAL) hinter Extbase zu setzen ist ein ganz anderes Abstraktionslevel.

Benjamin Eberlei per E-Mail vom 17.12.13 00:12

Zum jetzigen Zeitpunkt wird die DBAL in TYPO3 durch eine Systemextension [Glossar-eintrag] bereitstellt, die auf der externen Bibliothek AdoDB basiert, welche jedoch Anzeichen des Stillstands aufzeigt und davon ausgegangen werden kann, dass das Projekt nicht weiterentwickelt wird. [Linkt zu SourceForge]

Anhand dieser Fakten wird ersichtlich, dass die Integration von Doctrine erstrebenswert ist, da dadurch die Abhängigkeit zu dem inaktiven Projekt AdoDB aufgelöst werden kann. Da jedoch eine Integration von Doctrine ORM in Extbase nicht in der gegebenen Zeit, die für die Bearbeitung der Thesis zur Verfügung steht, zu realisieren ist, wurde der Fokus stattdessen auf die Integration von Doctrine CMS in TYPO3 gelegt, wodurch nicht nur Extbase von den Möglichkeiten eines DBAL profitieren kann, sondern der gesamte Core und somit alle Extensions die noch nicht mit Extbase erstellt worden sind.

Ferner wird durch diesen Ansatz eine stabile Basis zu Verfügung gestellt, auf der eine zukünftige Integration der ORM Komponente von Doctrine in Extbase aufbauen kann.

Ziel dieser Thesis ist es einen funktionierenden Prototypen zu entwickeln, der zum einen aus einer Extension besteht, die für die Integration von Doctrine DBAL zuständig ist und zum anderen aus einem modifizierten TYPO3, welches die neue API, die mit der Extension eingeführt wird, beispielhaft benutzt.

Im ersten Teil werden die eingesetzten Werkzeuge vorgestellt. Es wird erklärt warum diese und nicht andere eingesetzt worden sind und wie diese in Hinblick auf die Aufgabenstellung benutzt wurden.

Der zweite Teil beschreibt die praktische Umsetzung und schließt mit einer Demonstration wie der Prototyp getestet werden kann.

Teil drei gibt einen Ausblick auf die weitere Verwendung des Quellcodes und des Prototypen, während Teil vier mit einem Fazit schließt.

Im Anhang befindet sich neben den obligatorischen Verzeichnissen für Literatur und Abbildungen ein Glossar, sowie das Abkürzungsverzeichnis.

2.1 TYPO3 CMS

2.1.1 Geschichte

TYPO3 CMS ist ein Web Content Management-System (WCMS) und wurde von dem dänischen Programmierer Kaspar Skårhøj im Jahr 1997 zunächst für seine Kunden entwickelt - im Jahr 2000 von ihm unter der GNU General Public License v.2 (GPL2) veröffentlicht. Dadurch fand es weltweit Beachtung und erreichte eine breite Öffentlichkeit. Laut der Website T3Census¹ gab es am 7. April 2014 208561 Installationen von TYPO3 CMS.

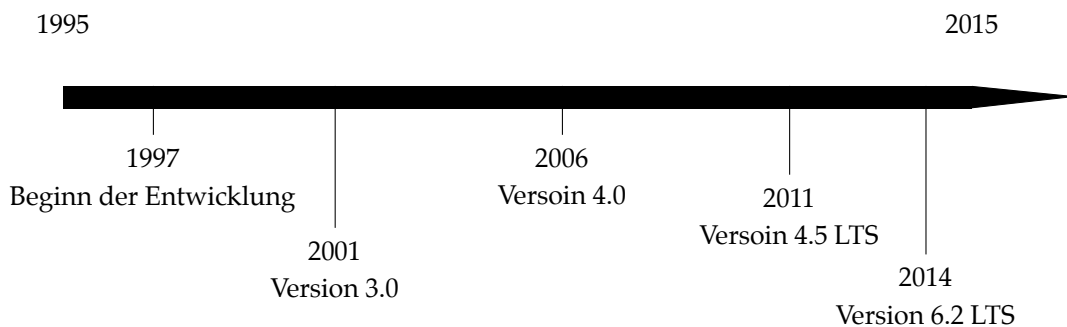


Abbildung 2.1: Zeitachse der TYPO3 CMS Entwicklung

Im Jahr 2012 entschied sich das Projekt zu einer Änderung in der Namesgebung:

- aus TYPO3 v4² wurde TYPO3 CMS
- aus FLOW3 wurde TYPO3 Flow
- und aus TYPO3 5.0 / TYPO3 Phoenix wurde TYPO3 Neos

Diese Änderung wurde notwendig, da schon länger abzusehen war, dass TYPO3 Phoenix nicht den Nachfolger von TYPO3 v4 darstellt. Somit war die Entwicklung von TYPO3 v4 in dem Versionszweig 4.x gefangen und es konnten keine neuen Features eingebaut oder veraltete Funktionen entfernt werden. Durch dieses neue Schema bekommt der Name "TYPO3" die Bedeutung einer Dachmarke zuteil, während "TYPO3 CMS", "TYPO3 Flow" und "TYPO3 Neos" Produkte innerhalb der TYPO3 Familie darstellen. Im weiteren Verlauf dieser Arbeit werden ausschließlich die neuen Namen verwendet.

¹ <http://t3census.info/>

² Damit ist das von Skårhøj entwickelte CMS gemeint, welches den 4.x Zweig des Projekts darstellt.

Heute kümmert sich ein Team um die Entwicklung von TYPO3 CMS und eines um TYPO3 Flow und TYPO3 Neos. Dahinter steht keine Firma, wie es bei anderen Open Source Projekten wie Drupal (Acquia) oder Wordpress (Automattic) vorzufinden ist, sondern die TYPO3 Association (T3Assoc). Die T3Assoc ist ein gemeinnütziger Verein und wurde 2004 von Kaspar Skårhøj und anderen Entwicklern gegründet um als Anlaufstelle für Spenden zu dienen, die die langfristige Entwicklung von TYPO3 sicherstellen sollen. Die Spenden werden in Form von Mitgliedsbeiträgen erhoben.³

2.1.2 Definition

TYPO3 CMS ist ein klassisches CMS, welches auf die Erstellung, die Bearbeitung und das Publizieren von Inhalten im Intra- oder Internet spezialisiert ist und es somit per Definition zu einem WCMS macht.

Daneben findet man auch die Bezeichnung Enterprise Content Management-System (ECMS)⁴, was als Hinweis auf den Einsatz des Systems für mittel- bis große Webprojekte dient.

2.1.3 Architektur und Aufbau von TYPO3 CMS

Im folgenden werden die grundlegenden Konzepte von TYPO3 CMS vorgestellt. Dort wo es für das weitere Verständnis notwendig ist, wird tiefer in das Thema eingestiegen. Ansonsten werden die Konzepte lediglich angerissen um einen generellen Überblick zu erhalten.

Webstack als Basis

TYPO3 CMS wurde in PHP: Hypertext Processor (PHP) - basierend auf dem Konzept der Objektorientierung - geschrieben und ist damit auf jeder Plattform lauffähig, die über einem PHP Interpreter verfügt. Die Version 6.2 von TYPO3 CMS benötigt mindestens PHP 5.3.7.

PHP bildet zusammen mit einem Apache Webserver und einer MySQL Datenbank den sogenannten Webstack, der abhängig von dem eingesetzten Betriebssystem MAMP (OSX / Mac), LAMP (Linux) oder WAMP (Windows) heißt.

In der Standardeinstellung kommt MySQL als Datenbank zum Einsatz - durch die Systemextension⁵ *DBAL* können jedoch auch Datenbanken anderer Hersteller angesprochen werden. Eine genaue Analyse dieser Extension erfolgt im Kapitel ??[Kapitel zur Analyse von ext:DBAL einfügen].

Ansichtssache

Aus Anwendersicht teilt sich TYPO3 CMS in zwei Bereiche:

³ <http://association.typo3.org/>

⁴ <http://www.typo3.org>

⁵ Eine kurze Einführung in die verschiedenen Arten von Extension findet sich im Kapitel 2.1.4

- das Backend
stellt die Administrationsoberfläche dar. Hier erstellen und verändern Redaktue-re die Inhalte; während Administratoren das System von hier aus konfigurieren
- das Frontend
stellt die Website dar, die ein Besucher zu Gesicht bekommt.

(vgl. [DRB08, S. 5])

Der Systemkern und die APIs

TYPO3 CMS besteht aus einem Systemkern, der lediglich grundlegende Funktionen zur Datenbank-, Datei- und Benutzerverwaltung zu Verfügung stellt. Dieser Kern ist nicht monolithisch aufgebaut, sondern besteht aus Systemextensions. (vgl. [Lab+06, S. 32])

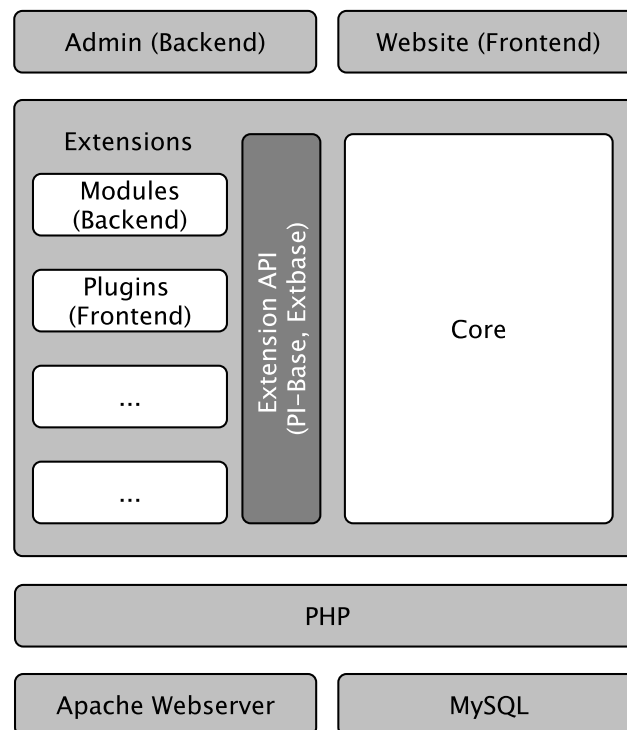


Abbildung 2.2: Schematischer Aufbau von TYPO3

Die Gesamtheit aller von TYPO3 CMS zur Verfügung gestellten APIs, wird als die *TYPO3 API* bezeichnet. Diese kann - analog zum Konzept von Backend und Frontend - in eine *Backend API* und eine *Frontend API* unterteilt werden kann. Die Aufgabe der Frontend API ist die Zusammenführung der getrennt vorliegenden Bestandteile (Inhalt, Struktur und Layout) aus der Datenbank oder dem Cache zu einer HTML-Seite. Die Backend API stellt Funktionen zur Erstellung und Bearbeitung von Inhalten zur Verfügung. (vgl. [DRB08, S. 5 ff.])

Die APIs, die keiner der beiden Kategorien zugeordnet werden kann, bezeichnet Dulepov [DRB08, S. 5 ff.] als *Common-API*. Die Funktionen der Common-API werden von allen anderen APIs genutzt. Ein Beispiel dafür stellt die Datenbank API dar, welche in der Regel nur einfache Funktionen wie das Erstellen, Einfügen, Aktualisieren, Lö-

schen und Leeren⁶ von Datensätzen bereitzustellen hat. Würde man je eine Datenbank API für das Frontend und das Backend zur Verfügung stellen, bricht man eine wichtige Regel der Objekt-orientierten Programmierung - Don't repeat yourself. Dieser - mit hoher Wahrscheinlichkeit - redundante Code würde die Wartbarkeit des Programms verschlechtern und die Fehleranfälligkeit erhöhen.

Auf die aktuelle Datenbank-API wird in Kapitel 2.4 näher eingegangen.

Verzeichnisstruktur

Im Gegensatz zu früheren TYPO3 CMS Versionen gibt es kein *Dummy-Package*⁷ mehr. Ab Version 6.2 enthält der Download lediglich den TYPO3 CMS Kern in Form des Verzeichnisses `typo3/`.

XCLASS

TYPO3 CMS besitzt einen Mechanismus, der es erlaubt Klassen zu erweitern oder Methoden mit eigenem Code zu überschreiben. Dies funktioniert für den Systemkern wie auch für andere Extensions. Dieses Feature nennt sich XCLASS und wird vom Prototypen eingesetzt um die Datenbankklasse von TYPO3 CMS zu überschreiben. Hier soll lediglich der Hintergrund zu XCLASS beschrieben werden.

Damit eine Klasse per XCLASS erweiterbar ist, darf sie nicht per `new()` Operator erzeugt werden, sondern mit der von TYPO3 CMS angebotenen Methode `\TYPO3\CMS\Core\Utility\GeneralUtility::makeInstance()`. Diese Methode sucht im globalen PHP-Array `$GLOBALS['TYPO3_CONF_VARS']['SYS']['Objects']` nach angemeldeten Klassen, instanziiert diese und liefert sie anstelle der Originalklasse zurück. Dieses Array dient der Verwaltung der zu überschreibenden Klassen und erfolgt in der Datei `ext_localconf.php` innerhalb des Extensionsverzeichnisses.

Der Mechanismus hat jedoch ein paar Einschränkungen:

- der Code der Originalklasse kann sich ändern. Es ist somit nicht sichergestellt, dass der überschreibende Code weiterhin das macht, wofür gedacht war
- XCLASSes funktionieren nicht mit statischen Klassen, statischen Methoden und finalen Klassen
- eine Originalklasse kann nur einmal per XCLASS überschrieben werden
- einige Klassen werden sehr früh bei der Initialisierung des System instanziiert. Das kann dazu führen, dass Klassen die als Singleton ausgeführt sind, nicht überschrieben werden können oder es kann zu unvorhergesehenen Nebeneffekten kommen.

2.1.4 Extensions

Extensions sind funktionale Erweiterungen. Sie interagieren mit dem Systemkern über die Extension API und stellen die Möglichkeit dar TYPO3 CMS zu erweitern und anzupassen.

⁶ CRUD - Create, Retrieve, Update und Delete

⁷ Damit ist ein weitgehend leeres Paket gemeint, dass alle Dateien enthält die im Webroot des Servers laufen sollen. Es stellt einen Container für die spätere Website dar.

Extensions werden - je nach Kontext - in unterschiedliche Kategorien eingeteilt, die hier kurz vorgestellt werden.

Einteilung

Systemextension werden mit dem System mitgeliefert und befinden sich ausschließlich im Ordner `typo3/sysext/`. Sie werden nochmals unterteilt in jene, die für den Betrieb von TYPO3 CMS unabdingbar sind und solche die nicht zwangsläufig installiert sein müssen, jedoch wichtige Funktionen beisteuern. Die Extension DBAL ist in die letzte Kategorie einzuordnen. Auf sie wird im Kapitel ?? näher eingegangen.

Neben Systemextensions gibt es noch globale⁸ und lokale Extensions. Lokale Extensions werden im Ordner `typo3conf/ext/` und globale Extensions im Ordner `typo3/ext` installiert.

Eine weitere Kategorisierung erfolgt nach dem Aufgabengebiet einer Extension. Die Festlegung auf eine der folgenden Kategorien hat keine direkte Auswirkung auf die Funktion der Extension. Sie wird von TYPO3 CMS hauptsächlich als Sortiermerkmal im Extension Manager (EM) genutzt.

Extension Manager

Der EM ist ein Backend (BE) Modul, über das die Extensions verwaltet werden können. Es erlaubt die Aktivierung, Deaktivierung, Herunterladen und das Löschen von Extensions. Darüberhinaus bietet der EM Möglichkeiten zur detaillierten Anzeige von Informationen über die Extensions wie das Changelog⁹, Angaben zu den Autoren und Ansicht der Dateien der Extension.

Verzeichnisstruktur

Unabhängig von der Einteilung der Extensions in die verschiedenen Kategorien unterscheiden sie sich nicht in der Verzeichnis- und Dateistruktur. Mit der Integration von Extbase in TYPO3 CMS hat sich eine neue Verzeichnisstruktur etabliert. Sie folgt dem Paradigma *Konvention statt Konfiguration*, was bedeutet, dass durch Einhaltung der Struktur keine weitere Konfiguration notwendig ist.

2.2 Doctrine

Das Doctrine Projekt besteht aus einer Reihe von PHP-Bibliotheken, die Schnittstellen rund um die Datenbankschicht bereitstellen. Die Konzepte sind beeinflusst von Javas *Hibernate*¹⁰ (vgl. [T3N09]) und dem Entwurfsmuster *Active Record*, welches von Martin Fowler [Fow03] vorgestellt wird.

⁸ Da globale Extensions nur in bestimmten Szenarien einen Sinn ergeben und in der Realität so gut wie nicht vorkommen, wird von der TYPO3 Community der Begriff "Extension" synonym zum Begriff "lokale Extension" verwendet. Die Arbeit folgt dieser Regelung.

⁹ Das Protokoll der Codeänderungen, die ein Programm im Laufe seines Lebens erlebt

¹⁰ <http://hibernate.org/>

Das Projekt wurde 2006 von Konsta Vesterinen initiiert¹¹ und im Jahr 2008 als Version 1.0.0 veröffentlicht. Die Version 2.0 wurde im Jahr 2010 unter dem neuen Projektleiter Benjamin Eberlei fertiggestellt.

Die beiden bekanntesten Produkte des Projekts sind:

- ORM - ermöglicht die objekrelationale Abbildung von Objekten auf Datenbanktabellen
- DBAL - stellt eine Datenbankabstraktionsschicht bereit

Wie aus Abbildung 2.3 ersichtlich wird, ist Doctrine DBAL lediglich als eine dünne Schicht auf Basis der PHP-Extension PHP Data Objects (PDO) ausgeführt, die die grundlegenden Funktionen zur Abstraktion von Datenbanken implementiert. Die ORM Schicht baut auf Doctrine DBAL auf.

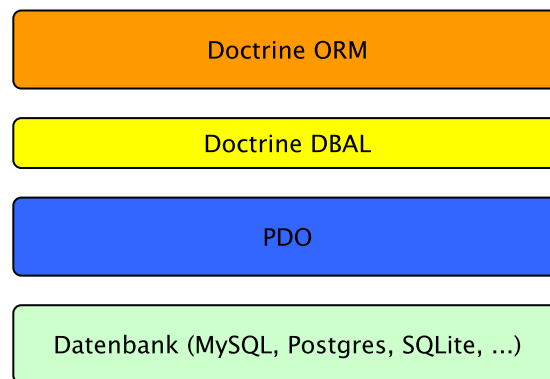


Abbildung 2.3: Schematischer Aufbau von Doctrine

2.2.1 ORM

“ Ein ORM ist eine Abstraktionsschicht zwischen relationaler Datenbank und der eigentlichen Anwendung. Statt per SQL kann man durch das ORM objektorientiert auf die Daten zugreifen.

Jonathan Wage [T3N09]

Das folgende Anwendungsbeispiel zeigt eine typische Situation. Es soll ein neuer Student in die Datenbank der Hochschule eingefügt werden. Im ersten Codelisting wird die Aufgabe auf dem herkömmlichen Weg gelöst. Dabei wird die Anfrage an eine MySQL und PostgreSQL Datenbank gesendet. Die Variable `$connection` enthält je eine initialisierte Verbindung zur entsprechenden Datenbank.

¹¹ <http://docs.doctrine-project.org/projects/doctrine1/en/latest/en/manual/acknowledgements.html>

```

1  <?php
2
3  $sql =
4      'INSERT INTO students ('first_name', 'last_name', 'enrolment_number')
5      VALUES ('Stefano', 'Kowalke', '12345');
6
7  // MySQLi
8  $result = mysqli_query($connection, $sql);
9
10 // PostgreSQL
11 $result = pg_query($connection, $sql);

```

Listing 1: Speichern eines Studenten in die Datenbank ohne ORM

Im folgenden Listing wird die gleiche Aufgabe mit Doctrine ORM gelöst. Es wird zunächst ein neues Objekt eines Studenten erzeugt und im weiteren Verlauf mit verschiedenen Daten angereichert. Anschließend wird es als zu speicherndes Objekt bei der Datenbank registriert und schließlich gespeichert.

```

1  <?php
2
3  $student = new Student();
4  $student->setFirstName('Stefano');
5  $student->setLastName('Kowalke');
6  $student->setEnrolmentNumber('12345');
7  $entityManager->persist($student);
8  $entityManager->flush();

```

Listing 2: Speichern eines Studenten in die Datenbank mit ORM

Der Code in Listing 2 gibt keinen Rückschluss auf die darunterliegende Datenbank. Die Daten des Studenten könnten in eine CSV-Datei, einer MySQL oder Postgres Datenbank gespeichert worden sein. Hingegen wurden in Listing 1 zwei verschiedene Methoden genutzt, um die Daten in eine MySQL und Postgres Datenbank zu schreiben. Die Speicherung in eine Textdatei wurde dabei nicht berücksichtigt.

Doctrine ORM ist für die Umwandlung des `$student`-Objekt in eine Structured Query Language (SQL)-Abfrage zuständig. Die erzeugte Abfrage ist mit der aus Codebeispiel 1 vergleichbar. Die Konvertierung der Anfrage in die verschiedenen SQL-Dialekte¹² erfolgt durch Doctrine DBAL.

2.2.2 DBAL

Doctrine konvertiert das Schema anhand von sehr unterschiedlichen Merkmalen in das SQL der jeweiligen Database Management System (DBMS), die zum Verständnis einen etwas tieferen Einstieg in die Eigenheiten der DBMS erfordern. Da dies den Umfang der Arbeit überschreitet, wurden markante Beispiele gewählt, die den Sachverhalt verdeutlichen.

¹² Als SQL Dialekt wird ein vom SQL-Standard abweichender Hersteller-spezifischer Sprachumfang bezeichnet. Ein Dialekt ist in der Regel kompatibel mit dem Standard und erweitert ihn um eigene Sprachkonstrukte.

Datenbankschemas werden in Doctrine von der Klasse `Schema` repräsentiert. Im Beispiel wird zunächst eine Instanz dieser Klasse erstellt und anschließend wird eine neue Tabelle und mehrere Tabellenspalten mit unterschiedlichen Datentypen angelegt. In Zeile 24 wird das Schema in eine SQL-Abfrage übersetzt. Davor existiert es lediglich als PHP-Objekt bis zum Ende der Laufzeit des Scripts.

Die Variable `$myPlatform` enthält die Information über das aktuelle benutzte DBMS.

```

1  <?php
2
3  $schema = new \Doctrine\DBAL\Schema\Schema();
4  $beUsers = $schema->createTable('be_users');
5  $beUsers->addColumn('uid', 'integer',
6      array('unsigned' => TRUE, 'notnull' => TRUE, 'autoincrement' => TRUE)
7  );
8  $beUsers->addColumn('pid', 'integer',
9      array('unsigned' => TRUE, 'default' => '0', 'notnull' => TRUE)
10 );
11 $beUsers->addColumn('username', 'string',
12     array('length' => 50, 'default' => '', 'notnull' => TRUE)
13 );
14 $beUsers->addColumn('password', 'string',
15     array('length' => 100, 'default' => '', 'notnull' => TRUE)
16 );
17 $beUsers->addColumn('admin', 'boolean',
18     array('default' => '0', 'notnull' => TRUE)
19 );
20 $beUsers->addColumn('history_data', 'text',
21     array('length' => 16777215, 'notnull' => FALSE)
22 );
23 $beUsers->addColumn('ses_data', 'text',
24     array('notnull' => FALSE)
25 );
26 $beUsers->setPrimaryKey(array('uid'));
27 $beUsers->addIndex(array('pid'), 'be_users_pid_idx');
28 $beUsers->addIndex(array('username'), 'be_users_username');
29
30 $queries = $schema->toSql($myPlatform);

```

Listing 3: Erstellen eines Schemas mit Doctrine

Die beiden Listings 4 und 5 zeigen den Inhalt von `$queries` – einmal für MySQL und für PostgreSQL.

```

1 // Der Inhalt von $queries für MySQL
2 CREATE TABLE `be_users` (
3     `uid` int(10) unsigned NOT NULL AUTO_INCREMENT,
4     `pid` int(10) unsigned NOT NULL DEFAULT '0',
5     `username` varchar(50) COLLATE utf8_unicode_ci NOT NULL DEFAULT '',
6     `password` varchar(100) COLLATE utf8_unicode_ci NOT NULL DEFAULT '',
7     `admin` tinyint(1) NOT NULL DEFAULT '0',
8     `history_data` mediumtext,
9     `ses_data` longtext,
10    PRIMARY KEY (`uid`),
11    KEY `be_users_pid_idx` (`pid`),
12    KEY `be_users_username` (`username`)
13 ) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

```

Listing 4: Das erstellte Schema als MySQL Anfrage

```

1 // Der Inhalt von $queries für PostgreSQL
2 CREATE TABLE be_users (
3     uid serial NOT NULL,
4     pid integer NOT NULL DEFAULT 0,
5     username character varying(50) NOT NULL DEFAULT ''::character varying,
6     password character varying(100) NOT NULL DEFAULT ''::character varying,
7     admin boolean NOT NULL DEFAULT false,
8     history_data text,
9     ses_data text,
10    CONSTRAINT be_users_pkey PRIMARY KEY (uid)
11 ) WITH (
12     OIDS=FALSE
13 );

```

Listing 5: Das erstellte Schema als PostgreSQL

Anhand der Beispiele können folgende Konvertierungen abgeleitet werden:

Abstraktion	MySQL	PostgreSQL
integer ... auto_increment	INT(10) ... AUTO_INCREMENT	serial
integer	INT(10)	INTEGER
string ... length => 50	VARCHAR(50)	CHARACTER VARYING(50)
boolean	TINYINT(1)	BOOLEAN
text ... length => 255	TINYTEXT	TEXT
text	LONGTEXT	TEXT

Tabelle 2.1: Typkonvertierung von Doctrine nach MySQL und PostgreSQL

Doctrine wählt die entsprechenden Typen anhand verschiedener Kriterien aus.

- Angabe von weiteren Optionen: `auto_increment`
- Angabe einer Länge: `length => 50` bzw. `length => 255`
- Konvertierung in äquivalente Datentypen: MySQL implementiert keinen `BOOLEAN` Typ, daher wird hier `TINYINT(1)` genutzt

Dabei ist zu beachten, dass die Werte in Klammern für String-Typen eine andere Bedeutung haben, als für numerische Datentypen. Wird ein `VARCHAR` mit der Länge 34 definiert, bedeutet dies, dass darin eine Zeichenkette mit maximal 34 Zeichen inklusive Leerzeichen gespeichert werden kann. Würde man auf den (vollkommen abwegigen) Gedanken kommen in dieser Spalte das Buch *The Hitchhiker's Guide to the Galaxy* von Douglas Adams speichern zu wollen, würde der Inhalt wie folgt aussehen: "Far out in the uncharted backwater" [Ada95, S. 3]. Der Rest des Textes wird abgeschnitten.

Bei numerischen Datentypen beschreibt der Wert allerdings die Anzeigenbreite - also die Anzahl der angezeigten Ziffern des gespeicherten Wertes. Sollte der in der Spalte gespeicherte Wert kleiner sein als die Länge der Anzeigenbreite, werden die restlichen Stellen nach links mit Leerzeichen aufgefüllt. Wurde die Option `ZEROFILL` gesetzt, werden die restlichen Stellen mit Nullen anstelle von Leerzeichen aufgefüllt. Der Wert beeinflusst in keinsten Weise den maximal speicherbaren Wert. In einer als `TINYINT` definierten Spalte können immer 256 Werte gespeichert werden. Unabhängig davon ob sie als `TINYINT(1)` oder `TINYINT(4)` deklariert wurde.

Doctrine spiegelt dieses Verhalten wider, indem die Angabe von `length` bei der Deklaration eines Integers dem Wert der Anzeigenbreite entspricht; bei String-Typen den tatsächlichen speicherbaren Wertebereich und bei der Deklaration eines Text-Datentyps als Auswahlkriterium in Bezug auf die unterschiedlichen `*TEXT` Datentypen.

2.2.3 Unterstützte Datenbank-Plattformen

Doctrine unterstützt die folgenden Plattformen und kann um weitere Datenbanken erweitert werden.

- DB2
- Drizzle
- MySQL
- Oracle
- PostgreSQL
- SQLAnywhere
- SQLite
- SQLAzure
- SQL Server

2.2.4 Konzepte und Architektur von Doctrine

Wie aus Abbildung 2.3 ersichtlich ist und schon erwähnt wurde, setzt Doctrine DBAL auf PDO auf. Im Grunde bildet Doctrine DBAL lediglich eine dünne Schicht auf PDO, während PDO selbst die Datenbankabstraktion und eine einheitliche API für verschiedene DBMS zur Verfügung stellt.

PDO besteht aus der Klasse `PDO`, die eine Verbindung zur Datenbank darstellt, und aus der Klasse `PDOStatement`, die zum einen ein (Prepared) Statement als auch das Ergebnis einer Datenbankabfrage repräsentiert. Abbildung 2.4 zeigt die Klassen mit einer Auswahl an Methoden.

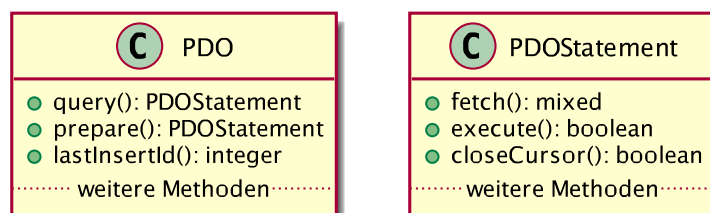


Abbildung 2.4: Die beiden PDO-Klassen

Die äquivalenten Klassen in Doctrine DBAL dazu sind `Doctrine\DBAL\Connection` und `Doctrine\DBAL\Statement`.

Die Integration der beiden PDO-Klassen in Doctrine DBAL erfolgt über das Adapter Entwurfsmuster (siehe Abbildung 2.5). Das hat den Vorteil, dass Doctrine DBAL nicht auf PDO als Datenbanktreiber festgelegt ist, sondern um weitere Treiber erweitert werden kann. Neue Treiber müssen lediglich das jeweilige Interface implementieren.

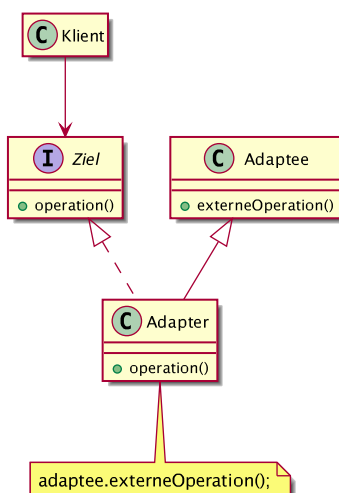


Abbildung 2.5: Das Adapter Entwurfsmuster in UML-Notation

Ein Adapter hat die Aufgabe eine Schnittstelle in eine Andere zu konvertieren. Ein Reisestecker oder ein Displayport-zu-VGA-Adapter stellen typische Beispiele aus der realen Welt dar.

In der Softwareentwicklung wird ein Adapter genutzt wenn eine externe Bibliothek in das eigene Projekt eingebunden werden soll und weder die API des eigenen Projekts noch die der externen Bibliothek verändert werden kann.

Die Akteure innerhalb des Adapter Entwurfsmusters sind:

- ein Klient: Stellt die Klasse dar, die die API nutzt. Dazu enthält sie eine interne Variable, die ein Objekt vom Typ des Interfaces erwartet.
- ein Ziel: ist ein Interface, welches die zu adaptierenden Methoden definiert. Der Klient wird gegen dieses Interface programmiert.

- ein Adapter: Stellt eine konkrete Implementierung des Interfaces dar und ist eine Kindklasse des Adaptee. Die Methoden des Adapters rufen intern die Methoden der Elternklasse auf, entsprechen nach außen jedoch der vom Klienten erwarteten API
- ein Adaptee: Die zu adaptierende externe Schnittstelle.

Das UML Diagramm in Abbildung 2.6 zeigt die Implementation der Connection API, die diesem Muster folgt.

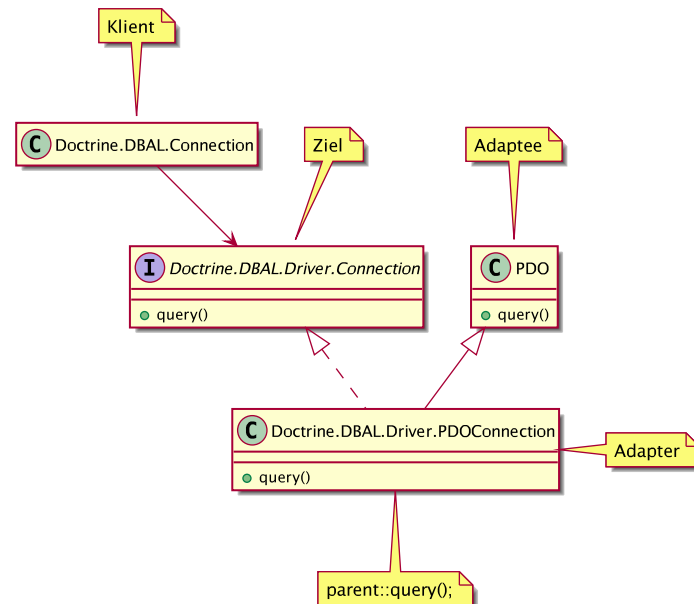


Abbildung 2.6: Aufbau des Connection-Objekts von Doctrine DBAL

Die Klasse `Doctrine\DBAL\Connection` stellt die Wrapper-Klasse beziehungsweise den Klienten dar. In der geschützten Variable `$_conn` wird ein Objekt erwartet, welches das Interface `Doctrine\DBAL\Driver\Connection` erfüllt. Bei PDO-basierten Verbindungen ist `Doctrine\DBAL\Driver\PDOConnection` ein konkretes Objekt dieses Interfaces. Es erbt außerdem von `PDO` und stellt den Adapter dar.

Die Klasse `PDOStatement` von `PDO` wird ebenfalls auf diese Art in Doctrine DBAL eingebunden.

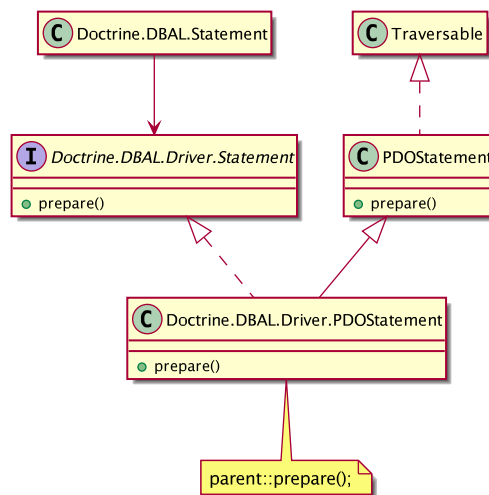


Abbildung 2.7: Aufbau des Statement-Objekts von Doctrine DBAL

Verbindung aufbauen

Eine Verbindung wird in Doctrine DBAL über `Doctrine\DBAL\DriverManager::getConnection()` angefordert.

```

1  $config = new \Doctrine\DBAL\Configuration();
2
3  $connectionParams = array(
4      'dbname' => 'hogwartsDB',
5      'user' => 'snape',
6      'password' => 'secret',
7      'host' => 'localhost',
8      'driver' => 'pdo_mysql',
9  );
10
11 $connection = \Doctrine\DBAL\DriverManager::getConnection($connectionParams, $config);

```

Listing 6

Anhand des angegebenen Wertes in `'driver'` wählt Doctrine den entsprechenden Datenbanktreiber aus und erstellt ein `PDOConnection`-Objekt, da es sich bei `'pdo_mysql'` um eine PDO-basierte Verbindung handelt.

Über den zweiten Parameter `$config` kann eine Konfiguration an Doctrine DBAL übergeben werden. Außerdem ist es möglich ein Loggerobjekt zu definieren, welches über dieses Objekt in Doctrine DBAL injiziert wird.

Einfache Datenbankabfragen

Im Folgenden wird die Verwendung von Doctrine DBAL an einfachen Beispielen erläutert. Als Grundlage der Abfragen und Ergebnisse dient eine Datenbank mit der folgenden Tabelle. Um die Beispiele einfach zu halten wurde auf eine korrekte Normalisierung der Tabelle, bei der die letzte Spalte in eine eigene Tabelle überführt und per Fremdschlüssel referenziert wird, verzichtet.

students				
id	first_name	last_name	house	
1	Lucius	Malfoy	Slytherin	
3	Herminone	Granger	Gryffindor	
4	Ronald	Weasley	Gryffindor	
5	Luna	Lovegood	Ravenclaw	
6	Cedric	Diggory	Hufflepuff	

Die als Beispiel dienende Abfrage soll die Nachnamen aller Studierenden in alphabetischer Reihenfolge ausgeben. Die in einer Variablen gespeicherten SQL-Abfrage wird an die Datenbank gesendet und das Ergebnis über eine Schleife ausgegeben. Zunächst wird der althergebrachte Weg gezeigt. Beide PHP-Extensions stellen dafür `*_query` Funktionen zur Verfügung, die eine Kennung der Datenbankverbindung zurückgeben. Im Fall eines Fehlers geben sie `FALSE` zurück.

```

1  $sql = 'SELECT last_name FROM students ORDER BY last_name';
2  // For MySQLi:
3  $result = mysqli_query($connection, $query);
4  while($row = mysqli_fetch_assoc($result)) {
5      echo $row['last_name'] . ' ';
6  }
7
8  // PostgreSQL:
9  $result = pg_query($query);
10 while($row = pg_fetch_assoc($result)) {
11     echo $row['last_name'] . ' ';
12 }

```

Listing 7

In Doctrine DBAL enthält `Doctrine\DBAL\Connection` die Verbindung zur Datenbank. Um eine Anfrage an die Datenbank zu senden, bietet die Klasse die Methode `query()` an. Die Methode gibt ein Objekt vom Typ `Doctrine\DBAL\Statement` zurück, dass das `IteratorAggregate` Interface implementiert. Dadurch kann über das Objekt mittels einer For-Schleife iteriert werden.

```

1  $sql = 'SELECT last_name FROM students ORDER BY last_name';
2
3  $statement = $connection->query($sql);
4
5  foreach($statement as $row) {
6      echo $row['last_name'] . ' ';
7  }

```

Listing 8

Die Ausgabe aller Beispiele lautet:

Diggory Granger Lovegood Malfoy Weasley

Funktionen auf der Ergebnismenge

Um das Ergebnis der Abfrage sinnvoll nutzen zu können, gibt es verschiedene Formate (engl. fetch styles) in denen das Ergebnis ausgegeben werden kann. In dem Beispielcode 8 wird die Standardeinstellung `PDO::FETCH_BOTH` genutzt. Dabei kann auf die Werte sowohl über einen Index als auch über den Spaltenbezeichner zugegriffen werden. Die interne Struktur sieht wie folgt aus:

```
1 foreach($statement as $row) {
2     print_r($row);
3 }
```

Listing 9

```
Array
(
    [last_name] => Diggory
    [0] => Diggory
)
Array
(
    [last_name] => Granger
    [0] => Granger
)
Array
(
    [last_name] => Lovegood
    [0] => Lovegood
)
...
```

Die Formatierung der Ergebnismenge wird über Konstanten gesteuert, die von PDO definiert und an die Methode `query()` als optionales Argument übergeben werden können. Eine andere Konstante ist `PDO::FETCH_NUM`, die das Ergebnis in ein Index-basiertes Array formatiert.

```
1 $statement = $connection->query($sql, PDO::FETCH_NUM);
2
3 foreach($statement as $row) {
4     echo $row[0];
5 }
```

In PDO existiert für jede Fetch-Methode aus der traditionellen Datenbankprogrammierung ein Äquivalent in Form einer Konstante:

- `PDO::FETCH_ASSOC = *_fetch_assoc()`¹³
- `PDO::FETCH_NUM = *_fetch_array()`
- `PDO::FETCH_ROW = *_fetch_row()`

¹³ Der Stern (*) dient als Platzhalter und kann wahlweise durch `mysql`, `mysqli` oder `pg` ersetzt werden.

Darüberhinaus definiert PDO noch weitere Konstanten, die keine Entsprechung haben:

- `PDO::FETCH_OBJ` - liefert jede Zeile der Ergebnisrelation als Objekt zurück. Die Spaltenbezeichner werden dabei zu Eigenschaften der Klasse.
- `PDO::FETCH_LAZY` - wie `PDO::FETCH_OBJ`. Das Objekt wird jedoch erst dann erstellt, wenn darauf zugegriffen wird.
- `PDO::FETCH_CLASS` - liefert eine neue Instanz der angeforderten Klasse zurück. Die Spaltenbezeichner werden dabei zu Eigenschaften der Klasse.
- `PDO::FETCH_COLUMN` - liefert nur eine Spalte aus der Ergebnismenge zurück.

Dies stellt eine nicht abschließende Aufzählung dar. Die Dokumentation von PDO benennt weitere Konstanten¹⁴, die für diese Arbeit jedoch nicht von Interesse sind.

Über das `Doctrine\DBAL\Statement`¹⁵ werden weitere Möglichkeiten wie die Methoden `Doctrine\DBAL\Statement::fetch()` und `Doctrine\DBAL\Statement::fetchAll()` angeboten, um das Ergebnis zu erhalten. Diese Methoden werden in Verbindung mit Prepared Statements genutzt, oder wenn statt der `foreach`-Schleife eine `while`-Schleife genutzt wird.

```
1 $statement = $connection->query($sql);
2
3 while($row = $statement->fetch(PDO::FETCH_ASSOC)) {
4     echo $row['last_name'];
5 }
```

Zudem gibt es mit `Doctrine\DBAL\Statement::fetch_column()` und `Doctrine\DBAL\Statement::fetch_object()` Alternativen zu den Konstanten.

Prepared Statements

Prepared Statements wurden bereits von der Datenbankerweiterung MySQLi eingeführt und sind keine Neuheit in der PHP Welt. Während MySQLi nur einen Typ der Prepared Statements unterstützt, stellt PDO eine weitere Variante zur Verfügung. Mit Doctrine DBAL können beide Ansätze genutzt werden.

Prepared Statements stellen normalen SQL-Code dar, bei dem die variablen Teile durch Platzhalter ersetzt wurden. Sie können als eine Vorlage für SQL-Abfragen verstanden werden, die mit verschiedenen Werten immer wieder ausgeführt werden sollen.

Ein Prepared Statements wird zunächst an die Datenbank gesendet, wo der SQL-Parser die Struktur des Templates einmalig analysieren und vorkompiliert im Cache speichern

¹⁴ <http://mx2.php.net/manual/en/pdo.constants.php>

¹⁵ Leider ist der Begriff dieser Klasse etwas unglücklich gewählt oder es ist ein Designfehler von PDO, denn ein Objekt dieser Klasse repräsentiert zum einen ein (Prepared) Statement und, nachdem die Anfrage ausgeführt wurde, die Ergebnisrelation. Die Methoden der Klasse agieren somit einmal auf dem Statement und einmal auf dem Ergebnis. Dies widerspricht dem Konzept in der Objekt-orientierten Programmierung, dass eine Klasse nur ein Verantwortlichkeit (engl. Single Responsibility Principle) haben darf. (vgl. [Mar08, S. 181])

kann. In einer zweiten Anfrage werden der Datenbank die Werte übermittelt, die bei jedem erneuten Aufruf vom Parser anstelle der Platzhalter eingesetzt werden. Dies macht zum einen die Ausführung der Anfrage schneller (vgl. [Pop07, S. 75]) und erhöht zudem die Sicherheit, da es SQL-Injections nahezu unmöglich macht. Mehr zu SQL-Injections in Kapitel 2.2.4.

Zur Demonstration soll je ein Codebeispiel dienen. Dabei werden neue Studierende in die oben gezeigte Datenbanktabelle eingefügt. Der sprechende Hut¹⁶ hat bereits über die Häuser der Neuzugänge entschieden.

Die Daten der Studierenden liegen in einem assoziativen Array vor und können somit über eine For-Schleife durchmustert werden. Pro Schleifendurchlauf wird ein Studierender der Datenbank hinzugefügt. Die Werte werden durch `Doctrine\DBAL\Connection::quote()` maskiert, um SQL-Injections zu unterbinden.

```

1  $students = array (
2      array (
3          'last_name' => 'Ellesmere',
4          'first_name' => 'Corin',
5          'house' => 1
6      ),
7      array (
8          'last_name' => 'Tugwood',
9          'first_name' => 'Havelock',
10         'house' => 4
11     ),
12     array (
13         'last_name' => 'Fenetre',
14         'first_name' => 'Valentine',
15         'house' => 3
16     )
17 )
18
19 foreach ($students as $student) {
20     $sql = 'INSERT INTO students (last_name, first_name, house)
21         VALUES (' . $connection->quote($student['last_name']) .
22             ', ' . $connection->quote($student['first_name']) .
23             ', ' . $connection->quote($student['house']) . ')';
24
25     $connection->query($sql);
26 }
```

Listing 10

Bei jedem Durchlauf wird eine neue Abfrage mit den aktuellen Daten erzeugt und an die Datenbank geschickt. In diesen Fall bietet sich die Benutzung von Prepared Statements an, da sich pro Iteration lediglich die Werte ändern.

¹⁶ http://de.harry-potter.wikia.com/wiki/Sprechender_Hut

```

1  $statement = $connection->prepare(
2      'INSERT INTO students (last_name, first_name, house)
3          VALUES (?, ?, ?)');
4
5  foreach ($students as $student) {
6      $statement->execute(
7          array(
8              $student['last_name'],
9              $student['first_name'],
10             $student['house']);
11      );
12  }

```

Listing 11

Die hier, anstelle der eigentlichen Daten, verwendeten Fragezeichen stellen die Platzhalter dar, die als *Positional Placeholders* bezeichnet werden. Die Daten werden der Methode `Doctrine\DBAL\Statement::execute()` in einem Array übergeben. Dabei ist die Reihenfolge wichtig, da ansonsten die Daten in die falschen Spalten der Tabelle geschrieben werden. Bei der Benutzung von Prepared Statements kann auf die Maskierung durch `Doctrine\DBAL\Connection::quote()` verzichtet werden, da dies die Datenbank übernimmt.

PDO bietet - im Gegensatz zu MySQLi - mit den *Named Paramentern* noch eine weitere Möglichkeit für Platzhalter an. Anstelle von Fragezeichen werden Bezeichner mit einem vorangestellten Doppelpunkt verwendet. Der Vorteil dieser Variante ist, dass die Reihenfolge bei der Übergabe der Daten an `Doctrine\DBAL\Statement::execute()` keine Rolle mehr spielt. Das folgende Listing zeigt den gleichen Code aus 11 jedoch diesmal mit Named Parametern. Die Daten werden dieses Mal als Key/Value-Paar übergeben, bei dem der Key den benannten Platzhalter darstellt und der Value die einzufügenden Werte.

```

1  $statement = $connection->prepare(
2      'INSERT INTO students (last_name, first_name, house)
3          VALUES (:lastname, :firstname, :house)');
4
5  foreach ($students as $student) {
6      $statement->execute(
7          array(
8              ':firstname' => $student['first_name'],
9              ':lastname'  => $student['last_name'],
10             ':house'     => $student['house']);
11      );
12  }

```

Trotz der veränderten Reihenfolge der Werte in dem Array werden sie in die richtige Spalte eingetragen.

Binding

Die Zuordnung einer Variablen zu einem Platzhalter wird *Binding* genannt - gebundene Variablen werden demzufolge als *Bounded Variables* bezeichnet.

Neben der gezeigten Bindung über `PDOStatement::execute()` bietet PDO spezialisierte Methoden an, was folgende Ursachen hat:

1. Bei der gezeigten Bindung werden die Variablen stets als String behandelt. Es ist nicht möglich dem DBMS mitzuteilen, das der übergebene Wert einem anderen Datentyp entspricht.
2. Die Variablen werden bei dieser Methode stets als In-Parameter übergeben. Auf den Wert der Variablen kann innerhalb der Funktion nur lesend zugegriffen werden. Man nennt diese Übergabe auch *by Value*. Es gibt jedoch Szenarien in denen der Wert der Variable innerhalb der Funktion geändert werden soll. Dann müssen die Parameter als Referenz (*by Reference*) übergeben werden und agieren als In/Out-Parameter. Einige DBMS unterstützen dieses Vorgehen und speichern das Ergebnis der Abfrage wieder in der übergebenen Variable.

Das Äquivalent zum obigen Beispiel ist `Doctrine\DBAL\Statement::bindValue()`, bei der die Variable als In-Parameter übergeben wird. Für jeden zu bindenden Platzhalter muß die Methode aufgerufen werden, die dessen Bezeichner, den zu bindenden Wert und die optionale Angabe des Datentyps erwartet. Der Datentyp wird der Datenbank über PDO-Konstanten mitgeteilt. `PDO::PARAM_STR` ist der Standardwert des zweiten Parameters.

```

1  $statement = $connection->prepare(
2      'INSERT INTO students (last_name, first_name, house)
3          VALUES (:lastname, :firstname, :house)');
4
5  foreach ($students as $student) {
6      $statement->bindValue(':lastname', $student['first_name']);
7      $statement->bindValue(':firstname', $student['last_name']);
8      $statement->bindValue(':house', $student['house'], PDO::PARAM_INT);
9
10     $statement->execute();
11 }

```

Listing 12

`Doctrine\DBAL\Statement::bindParam()` ist die Methode zur Übergabe der zu bindenden Werte per Referenz und unterscheidet sich von `Doctrine\DBAL\Statement::bindValue()` grundlegend. Der in der Variable gespeicherte Wert wird erst dann aus dem Speicher ausgelesen, wenn `Doctrine\DBAL\Statement::execute()` ausgeführt wird. Im Vergleich dazu wird der Wert schon bei dem Aufruf von `Doctrine\DBAL\Statement::bindValue()` vom SQL-Parser ausgelesen und in das Prepared Statement eingesetzt. Aus diesem Grund muß `Doctrine\DBAL\Statement::bindValue()` innerhalb der Schleife aufgerufen werden.

```

1  $statement = $connection->prepare(
2      'INSERT INTO students (last_name, first_name, house)
3          VALUES (:lastname, :firstname, :house)');
4
5  $statement->bindParam(':lastname', $student['first_name']);
6  $statement->bindParam(':firstname', $student['last_name']);
7  $statement->bindParam(':house', $student['house'], PDO::PARAM_INT);
8
9  foreach ($students as $student) {
10     $statement->execute();
11 }

```

SQL-Injections

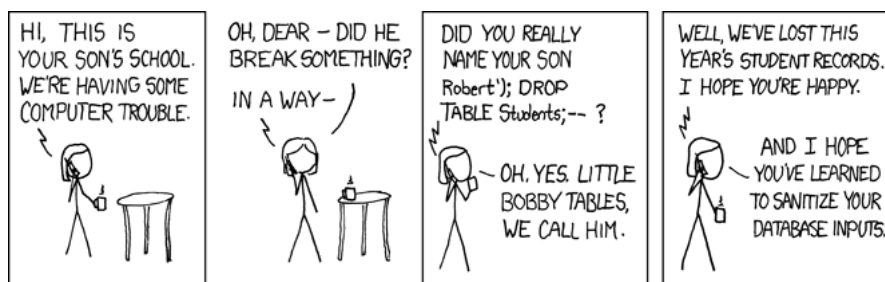


Abbildung 2.8: xkcd: Exploits of a mom¹⁷

Bei SQL-Injections kann über das Frontend einer Anwendung eine Zeichenkette in eine SQL-Abfrage injiziert werden, die die Fähigkeit besitzt den betroffenen SQL-Code derart zu verändern, dass er

- Informationen wie den Adminbenutzer der Webanwendung zurückliefert
- Daten in der Datenbank manipuliert um ein neuer Adminbenutzer anzulegen
- oder die Datenbank ganz- oder teilweise löscht

Für ein kurzes Beispiel einer SQL-Injection soll ein Formular dienen, indem nach den Nachnamen der Studierenden aus Hogwarts gesucht werden kann. Der gesuchte Datensatz wird ausgegeben wenn er gefunden wird, ansonsten erscheint eine entsprechende Meldung das nichts gefunden wurde. Der in das Inputfeld eingegebene Wert wird von PHP automatisch in der Variablen `$_REQUEST` gespeichert und kann in der Anwendung ausgelesen werden. `'SELECT * FROM students WHERE last_name = Diggory'` stellt eine mögliche, zu erwartende SQL-Anfrage dar.

¹⁷ <http://xkcd.com/327/>

```

1  $sql = "SELECT * FROM students
2      WHERE last_name = '" . $_REQUEST['lastName'] . "'";
3
4  $statement = $connection->query($sql);
5
6  foreach($statement as $student) {
7      echo 'Lastname: ' . $student['last_name'] . "\n";
8      echo 'Firstname: ' . $student['first_name'] . "\n";
9      echo 'Haus: ' . $student['house'] . "\n";
10 }

```

Listing 13

Dieser Code beinhaltet zwei Fehler:

1. Es wird nicht überprüft, ob `$_REQUEST['lastName']` leer ist oder was ganz anders enthält als erwartet wie zum Beispiel ein Objekt oder Array.
2. die Benutzereingabe wird nicht maskiert

Im Falle einer leeren Variable, sähe die Abfrage so aus:

```
'SELECT * FROM students WHERE last_name = ';
```

Im besten Fall gibt sie eine leere Ergebnismenge zurück, im schlechtesten einen Fehler. Diese Problem ist leicht zu lösen, indem zum einen auf die Existenz der Variablen geprüft wird und zum anderen ob sie einen Wert enthält. Zusätzlich sollte noch auf den Datentyp des enthaltenen Wertes geprüft werden. Erst dann wird die Anfrage abgesetzt.

Da die Eingabe nicht maskiert wird, interpretiert der SQL-Parser einige Zeichen als Steuerzeichen der SQL-Syntax. Beispiele solcher Zeichen sind das Semikolon (;), der Apostroph ('), der Backslash (\) oder zwei Minuszeichen (--).

Selbst ein Websitebesucher ohne böse Ansichten könnte mit der Suche nach einem Studenten mit dem Namen *O'Hara* die SQL-Injection auslösen. Die in diesen Fall an die Datenbank gesendetet SQL-Anfrage `'SELECT * FROM students WHERE last_name = 'O'Hara'`; würde wohl einen Fehler auslösen, da der Parser die Anfrage nach dem *O* anhand des Apostroph als beendet interpretiert und *Hara* kein gültiges Sprachkonstrukt von SQL darstellt.

Ein Angreifer könnte hingegen die Eingabe in das Formular nach `' or '1'='1` verändern. Damit würde sich diese Abfrage

```
'SELECT * FROM students WHERE last_name = ' or '1'='1';
```

ergeben.

Wird die Maskierung mit einer entsprechenden Prüfung von Benutzereingaben kombiniert, verhindert das die Gefahr von SQL-Injections – eine richtige Anwendung vorausgesetzt.

Wie in Kapitel ?? bereits erwähnt wurde, müssen an `Doctrine\DBAL\Connection::query()` übergebene SQL-Anfragen mit `Doctrine\DBAL\Connection::quote()` maskiert werden. Traditionell wird dafür die PHP-Funktion `addslashes()` oder die jeweiligen Maskierungsmethoden der DBMS verwendet. Für MySQLi wird zum Beispiel `mysqli_real_escape_string()` benutzt.

Werden Prepared Statements genutzt, müssen die Benutzereingaben trotzdem überprüft, jedoch nicht mehr maskiert werden. Dies übernimmt der SQL-Parser, der legig-

lich die Werte in das vorkompilierte Prepared Statement einsetzt. Wird dem Prepared Statement noch der Typ des Wertes mitgeteilt, kann das DBMS eine Typprüfung vornehmen und ggf. einen Fehler zurückgeben.

Limitierungen

Bei einer Abstraktion wird stets etwas Spezifisches, durch das Weglassen von Details, in etwas Allgemeines überführt. Doctrine DBAL geht den umgekehrten Weg – es werden allgemeine SQL-Anfragen in den Dialekt¹⁸ des Herstellers übersetzt.

Aus diesem Grund vermag es PDO nicht eine SQL-Abfrage, die in dem Dialekt eines Herstellers formuliert wurde, in den eines anderen zu übersetzen.

Das folgende Beispiel zeigt die von MySQL unstützte vereinfachte Form eines `INSERT`-Statements. Dies stellt eine Abweichung vom SQL-Standard dar (vgl. [ISO92, S. 388]) und ist somit nicht portabel.

```
1 INSERT INTO students SET last_name='Kowalke', first_name='Stefano';
```

Listing 14

Stattdessen muss die Anfrage so nah wie möglich am Standard gestellt werden, um als portabel zu gelten.

```
1 INSERT INTO students (last_name, first_name) VALUES('Kowalke', 'Stefano');
```

Listing 15

2.3 Arbeitsweise

Aufgrund seiner Aufgabe stellt der Prototyp einen tiefen Eingriff in die Architektur von TYPO3 dar. Vor diesem Hintergrund ist es umungänglich, dass er alle Anforderungen erfüllt, die an eine Systemextension gestellt werden, auch wenn er keine Systemextension ist.

Dies fängt mit der Einhaltung der TYPO3 Coding Guidelines an, geht über die Versionierung des Codes hin zum Einbinden in das TYPO3 Testframework

2.3.1 Formatierung des Quellcodes

Programmierer neigen zu unterschiedlichen Programmierstilen, was solange kein Problem darstellt, solange sie allein an einem Projekt arbeiten. Kommen mehr Entwickler hinzu und es vermischen sich verschiedene Stile, kann es schnell unübersichtlich werden. Um den Sinn eines Programms zu verstehen, hilft es allmein, wenn die Codebasis in einer konsistenten Form formatiert wurde. Dies erhöht die Wartbarkeit und verbessert die Code Qualität.

¹⁸ Als Dialekt wird die Hersteller-eigene Implementation des SQL-Standards genannt, dass sich im Umfang und Syntax vom Standard unterscheidet.

Coding Guidelines stellen dabei das Regelwerk dar auf das sich die Entwickler eines Projekts geeingt haben.

In den TYPO3 Coding Guidelines (CGL)¹⁹ wird die Verzeichnisstruktur von TYPO3 selbst, sowie die von Extensions erläutert. Sie erklären die Namenskonventionen für Dateien, Klassen, Methoden und Variablen und beschreiben den Aufbau einer Klassen-datei mit notwendigen Inhalt, der unabhängig von dem Zweck der Klasse vorhanden sein muss.

Der größter Teil der CGL behandelt die Formatierung verschiedene Sprachkonstrukte wie Schleifen, Arrays und Bedingungen.

Da das Regelwerk mit 28 Seiten recht umfangreich aussfällt, wird zur Überprüfung des Quellcodes das Programm PHP_CodeSniffer²⁰ in Zusammenhang mit dem entsprechenden Regelset für das TYPO3 Projekt ²¹ verwendet.

2.3.2 Unit Testing

Eine Anforderung an den Prototyp war, dass er die alte Datenbank API weiterhin unterstützt. Dadurch ist gewährleistet, dass noch nicht angepasste Extensions weiterhin funktionieren. Um dies zu überprüfen muß die Funktionalität von TYPO3 in Verbindung alter API / neue API fortlaufend getestet werden.

Ein möglicher Ansatz wäre es, mit zwei identischen TYPO3 Installationen zu starten, die mit der Zeit auseinander divergieren, indem eine der beiden APIs immer mehr auf die neue API umgebaut würde. Das Testen dabei kann jedoch nur auf eine stichprobenartige Überprüfung der Funktionalität des manipulierten Systems beruhen, da es nahezu unmöglich ist, durch dieses manuelle Vorgehen alle Testfälle abzudecken.

Ein andere Ansatz würde auf der Code Ebene ansetzen und pro Methode der alten API verschiedene Testfälle definieren, welche nachvollziehbar wären und immer wieder ausgeführt werden könnten.

Das dafür in Frage kommende Framework heißt PHPUnit²². Es stellt das PHP Pedant von dem aus der Javawelt bekannten JUnit dar und wird von TYPO3 unterstützt. TYPO3 bringt selbst schon über 6000 UnitTests mit²³.

Das eingangs umrissene Szenario stellt lediglich ein greifbares Beispiel für den Nutzen von Unit Tests dar und ist keineswegs auf Spezialfälle wie Refactorings oder dem Austausch einer API beschränkt. In der vorliegenden Arbeit wurden alle implementierten Methoden der neuen API mit Unit Tests – in Verbindung mit PHPUnit – abgedeckt.

¹⁹ http://docs.typo3.org/typo3cms/CodingGuidelinesReference/6.2/_pdf/manual.t3cgl-6.2.pdf

²⁰ https://github.com/squizlabs/PHP_CodeSniffer

²¹ <https://github.com/typo3-ci/TYP03CMS>

²² <http://phpunit.de/>

²³ <https://travis-ci.org/TYP03/TYP03.CMS/builds/23070563>

2.3.3 Versionsverwaltung

Als Versionsverwaltung wurde GIT²⁴ in Verbindung mit dem Code Hostingdienst GitHub²⁵ genutzt. Github dient zum einen als Backup im Falle eines Festplattenausfalls und zum anderen als späterer Anlaufpunkt der Extension für Interessierte.

Ferner hat sich um GitHub eine Vielzahl von Services entwickelt, welche unter anderen dabei hilft, die CodeQualität zu analysieren. Die zwei zu erwähnenden Projekte sind Travis²⁶ und Scrutinizer²⁷, welche für das Projekt genutzt wurden.

2.3.4 Travis-CI

Travis-CI ist ein *Continuous Integration* (CI) Service, welcher auf Github gehostete baut.

Der Begriff "bauen" kommt von Sprachen wie C oder Java, die erst kompiliert werden müssen (engl. to compile oder to build), bevor sie ein ausführbares Programm darstellen. Im Gegensatz zu interpretierten Sprachen, die zur Laufzeit von einem Interpreter übersetzt werden. Hier definiert der Begriff "bauen" im Zusammenhang mit CI das auschecken des Codes aus einem Repository und die Ausführung von:

- Tests (Unit-, Smoke-, Akzeptanz- und Behaviortests),
- statischer Codeanalysen wie den oben genannten PHP_CodeSniffer, PHPDepend²⁸, PHP Mess Detection²⁹ oder PHP Copy and Paste³⁰

Die Konfiguration von Travis-CI erfolgt über eine Textdatei im YAML-Format³¹ und liegt im Wurzelverzeichnis des Projekts.

[Beispiel einer YAML Datei für Travis einfügen]

Da der Prototyp aus zwei Teilen besteht, wurden zwei Travisprojekte erstellt. Eins für die Extension und eins für den modifizierten TYPO3 Kern.

Konfiguration für die Extension

PHPUnit CodeSniffer PHPCPD PHPMessDetection PHPLOC

Ziel war eine möglichst 100% Testabdeckung zu erreichen.

Konfiguration für den TYPO3 Kern

Während bei den Tests der Extension auf eine innere Konsistenz des Programmcodes geschaut wurde, wurde bei den Tests des Kerns Augenmerk auf die Integration der

²⁴ <http://git-scm.com/>

²⁵ <http://github.com>

²⁶ LINK

²⁷ LINK

²⁸ <http://pdepend.org/>

²⁹ <http://phpmd.org/>

³⁰ <https://github.com/sebastianbergmann/phpcpd>

³¹ <http://www.yaml.org/start.html>

Extension in das modifizierte TYPO3 gelegt. Ziel war es, dass alle mitgelieferten Tests des Cores erfüllt werden.

2.3.5 IDE

Als Editor wurde die Integrated Development Environment (IDE) PHPStorm verwendet, die über Autovervollständig von Variablen, Methoden und Klassen, unterstützt den Programmierer bei der Erstellung von Klassen durch Templates und bietet vielseitige Integration von externen Programmen.

PHPStorm verfügt über einen Debug Listener, der auf ein vom Browser gesendetes Token wartet und bei Empfang den Debug Prozess startet. Für den verwendeten Browser *Chrome* ist ein Addon verfügbar, mittels diesem das Senden des Debug-Tokens per Knopfdruck ein- und ausgeschaltet werden kann. Wird der Browser und die IDE in den Debug-Modus gesetzt und TYPO3 CMS neugeladen, bleibt das Programm an dem gesetzten Breakpoint stehen.

2.4 Aktuelle Situation

In Kapitel 2.1.3 wurde bereits darauf hingewiesen, dass TYPO3 CMS eine einheitliche Datenbank API anbietet. Diese wird durch die Klasse `\TYPO3\CMS\Core\Database\DatabaseConnection` bereitgestellt und ist über die globale Variable `$GLOBALS['TYPO3_DB']` verfügbar. Sie bietet Methoden zum lesenden und schreibenden Zugriff auf die Datenbank an.

```

1  $GLOBALS['TYPO3_DB']->exec_UPDATEquery(
2      $this->user_table,
3      $this->userid_column . ' = ' .
4          $this->db->fullQuoteStr($tempuser[$this->userid_column], $this->user_table),
5      array($this->lastLogin_column => $GLOBALS['EXEC_TIME'])
6  );

```

Listing 16: Aktualisierung des Zeitpunkts des letzten Logins

Durch die Nutzung der Datenbank API wird zum einen die Integrität der Daten sichergestellt und zum anderen die Nutzung der PHP-Datenbankfunktionen abstrahiert. Somit kann die Implementation der Datenbank API-Methoden angepasst werden ohne die API selbst zu verändern. Die Umstellung von den MySQL- auf die MySQLi-Methoden ist ein Beispiel, wurde auf diese Weise durchgeführt.³²

Die Datenbank API bietet eine Vielzahl an Methoden, die sich in die folgenden fünf Gruppen einteilen lassen.

Die erste Gruppe besteht aus Methoden, die anhand der übergebenen Parametern einen SQL-Query generieren. Damit werden die typischen CRUD-Operationen 6) abgebildet.


 TYPO3\CMS\Core\Database\DatabaseConnection
<ul style="list-style-type: none"> ● <code>INSERTquery(\$table, \$fields_values, \$no_quote_fields = FALSE): string</code> ● <code>UPDATEquery(\$table, \$where, \$fields_values, \$no_quote_fields = FALSE): string</code> ● <code>DELETEquery(\$table, \$where): string</code> ● <code>SELECTquery(\$select_fields, \$from_table, \$where_clause, \$groupBy = "", \$orderBy = "", \$limit = ""): string</code> ● <code>TRUNCATEquery(\$table): string</code>

Abbildung 2.9: DatenbankConnection mit den generierenden Methoden

Folgendes Codebeispiel aus `\TYPO3\CMS\Core\Authentication\AbstractUserAuthentication` zeigt die Funktionsweise von `SELECTquery()`. Der Kommentar zeigt die generierte SQL-Query.

```

1  // DELETE FROM sys_file_reference WHERE tablenames='pages';
2  $deleteQuery = $GLOBALS['TYPO3_DB']->DELETEquery(
3      'sys_file_reference',
4      'tablenames=' . $GLOBALS['TYPO3_DB']->fullQuoteStr('pages', 'sys_file_reference')
5  );

```

Listing 17: Löschen eines Datensatzes aus einer Tabelle

Eine Ebene höher setzt die nächste Gruppe an, die die eben gezeigten Methoden nutzt und den generierten SQL-Query ausführt und eine Ergebnismenge vom Typ `mysqli_result` zurückliefert.

³² <http://bit.ly/typo3cms-switch-from-mysql-to-mysqli>


 TYPO3\CMS\Core\Database\DatabaseConnection
<ul style="list-style-type: none"> ● <code>exec_INSERTquery(\$table, \$fields_values, \$no_quote_fields = FALSE): mysqli_result object</code> ● <code>exec_UPDATEquery(\$table, \$where, \$fields_values, \$no_quote_fields = FALSE): mysqli_result object</code> ● <code>exec_SELECTquery(\$select_fields, \$from_table, \$where_clause, \$groupBy = "", \$orderBy = "", \$limit = ""): mysqli_result object</code> ● <code>exec_SELECT_mm_query(\$select, \$local_table, \$mm_table, \$foreign_table, \$whereClause = "", \$groupBy = "", \$orderBy = "", \$limit = "")</code> ● <code>exec_SELECT_queryArray(\$queryParts)</code> ● <code>exec_SELECTgetRows(\$select_fields, \$from_table, \$where_clause, \$groupBy = "", \$orderBy = "", \$limit = "", \$uidIndexField = "")</code> ● <code>exec_SELECTgetSingleRow(\$select_fields, \$from_table, \$where_clause, \$groupBy = "", \$orderBy = "", \$numIndex = FALSE)</code> ● <code>exec_SELECTcountRows(\$field, \$table, \$where = "")</code> ● <code>exec_TRUNCATEquery(\$table): mixed</code> ● <code>exec_DELETEquery(\$table, \$where): mysqli_result object</code>

Abbildung 2.10: DatenbankConnection mit den ausführenden Methoden

In der dritte Gruppe befinden sich die Methoden, die im weitesten Sinn zur Verarbeitung der Ergebnismenge genutzt werden. Darunter fallen

- jene die die Daten aus der Ergebnismenge extrahieren
- mie für die Fehlerbehandlung genutzt werden können,
- sowie Methoden die Informationen über die Ergebnismenge breitstellen


 TYPO3\CMS\Core\Database\DatabaseConnection
<ul style="list-style-type: none"> ● <code>sql_query(\$query): mysqli_result object boolean</code> ● <code>sql_error(): string</code> ● <code>sql_errno(): integer</code> ● <code>sql_num_rows(\$res): integer</code> ● <code>sql_fetch_assoc(\$res): array boolean</code> ● <code>sql_fetch_row(\$res): array boolean</code> ● <code>sql_free_result(\$res): boolean</code> ● <code>sql_insert_id(): integer</code> ● <code>sql_affected_rows(): integer</code> ● <code>sql_data_seek(\$res): boolean</code> ● <code>sql_field_types(\$res, \$pointer): string</code>

Abbildung 2.11: DatenbankConnection: Methoden zur Verarbeitung der Ergebnismen-ge

Die nächste Gruppe besteht aus Hilfsmethoden, die genutzt werden um

- einen SQL-Query an die Datenbank zu senden
- Benutzereingaben zu maskieren
- Listen von Integeren zu normalisieren
- eine WHERE-Bedingung aus Komma-separierten Datensätzen zu erzeugen

C	TYPO3\CMS\Core\Database\DatabaseConnection
<ul style="list-style-type: none">● fullQuoteStr(\$str, \$table, \$allowNull = FALSE): string● fullQuoteArray(\$arr, \$table, \$noQuote = FALSE, \$allowNull = FALSE): array● quoteStr(\$str, \$table): string● escapeStrForLike(\$str, \$table): string● cleanIntArray(\$arr): array● cleanIntList(\$list): string● stripOrderBy(\$str): string● stripGroupBy(\$str): string● splitGroupOrderLimit(\$str): array● getDateFormats(\$table): array● listQuery(\$field, \$value, \$table): string● searchQuery(\$searchWords, \$fields, \$table, \$constraint = self::AND_Constraint): string	

Abbildung 2.12: DatenbankConnection: Hilfsmethoden

Die letzte Gruppe besteht aus einer Reihe von Methoden, die verschiedene Metadaten über die Datenbank zur Verfügung stellen. Der Name impliziert, dass sie für Administrative Tätigkeiten genutzt werden, was jedoch irreführend ist. Sie werden hauptsächlich während der Installation vom *Installation Tool* verwendet, um Informationen über die zugrundeliegende Datenbank zu erhalten

C	TYPO3\CMS\Core\Database\DatabaseConnection
	<ul style="list-style-type: none">● admin_get_dbs(): array● admin_get_tables(): array● admin_get_fields(\$tableName): array● admin_get_keys(\$tableName): array● admin_get_charsets(): array● admin_query(\$query): mysqli_result object

Abbildung 2.13: DatenbankConnection mit administrativen Methoden

Aus historischen Gründen nutzt die Datenbank API die prozeduralen MySQLi Funktionen anstelle der Objekt-orientierten API von MySQLi. Dadurch stellt \TYPO3\CMS\Core\Database\DatabaseConnection kein Verbindungsobjekt dar - wie zum Beispiel PDO - sondern eine Sammlung an Datenbankfunktionen. Dadurch gibt es auch kein Objekt, welches die Ergebnismenge repräsentiert, was \TYPO3\CMS\Core\Database\DatabaseConnection zu einer recht große Klasse mit 1950 Zeilen Code und 76 Methoden.

Im Gegensatz dazu steht die Anzahl von 40 UnitTests mit 49 Assertions für diese Klasse, die lediglich ein paar Hilfsmethoden testen. Es ist somit nicht sichergestellt, ob die Methoden das tun, was sie vorgeben zu tun. In den letzten Monaten wurde jedoch ein Framework für Funktionale Tests in TYPO3 CMS integriert, so dass die Methoden immerhin indirekt getestet werden.

[Und dass obwohl TYPO3 CMS mit aktuell 5527 Unit Tests und 8667 Assertions schon eine recht gute Testabdeckung hat 27% der Dateien und 36% der abgedeckten Zeilen]

2.4.1 Prepared Statements

Seit TYPO3 CMS 4.5 können Prepared Statements für **SELECT** Anweisungen verwendet werden. TYPO3 CMS unterstützt sowohl *Positional Parameters* wie auch *Named Parameters*.

```

1  $statement = $GLOBALS['TYPO3_DB']->prepare_SELECTquery(
2      '*', 'bugs', 'reported_by = ? AND bug_status = ?'
3  );
4  $statement->execute(array('goofy', 'FIXED'));
5
6  $statement = $GLOBALS['TYPO3_DB']->prepare_SELECTquery(
7      '*', 'bugs', 'reported_by = :nickname AND bug_status = :status'
8  );
9  $statement->execute(array(':nickname' => 'goofy', ':status' => 'FIXED'));

```

Listing 18: Positional und Named Prepared Statements der TYPO3 CMS Datenbank API

\TYPO3\CMS\Core\Database\DatabaseConnection::prepare_SELECTquery liefert ein Objekt der Klasse \TYPO3\CMS\Core\Database\PreparedStatement zurück, welches sich an der API von PDO orientiert.

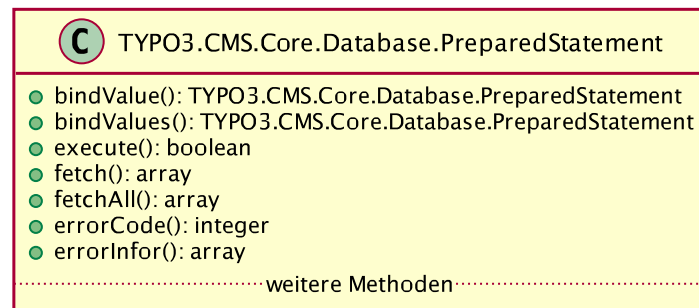


Abbildung 2.14: Die Klasse PreparedStatement mit ausgewählten Methoden

2.4.2 Datenbankschema

Nach der Installation von TYPO3 CMS beinhaltet die Datenbank rund 60 einzelne Tabellen. Die Anzahl hängt von der Installation der optionalen Systemextensions ab.

Wichtige Tabellen sind:

- pages – Enthält die Seiten
- tt_content – Enthält die Inhaltselemente, die auf den Seiten dargestellt werden
- be_groups / fe_groups – Enthält die Backend- beziehungsweise die Frontendgruppen
- be_users / fe_users – enthält die Backend- beziehungsweise die Frontendbenutzer

Dazu kommen Tabellen

- die gecachte Daten und Sessions beinhalten,
- die der Indexierung des Inhalts dienen
- sowie zum Protokollieren von Systemereignissen

Die Inhalte werden in TYPO3 CMS in einem Dateisystem ähnlichen Baumstruktur verwaltet. Eine Webseite wird darin durch einen Datensatz vom Typ *Seite* repräsentiert. Dieser Datensatz hat eine ID, die im gleichnamigen Feld in der Tabelle *pages* gespeichert wird. Diese ist der *Unique Identifier* des Datensatzes.

Die Inhalte einer Webseite wie Texte, Bilder oder Formulare werden innerhalb einer Seite abgelegt. TYPO3 CMS bietet hierzu eine breite Palette von verschiedenen Elementen an. Zudem können Plugins und wiederum Seiten innerhalb eines Seitendatensatzes abgelegt werden. Diese Liste kann unbegrenzt fortgeführt werden, da jede Extension neue Elemente einführen kann, welches in einer Seite ablegbar sind. Es ist lediglich wichtig zu wissen, dass Datensätze innerhalb von Seiten abgelegt werden können. Die Inhaltselemente werden hauptsächlich in der Tabelle *tt_content* gespeichert beziehungsweise in den Tabellen, die die Extension vorsieht.

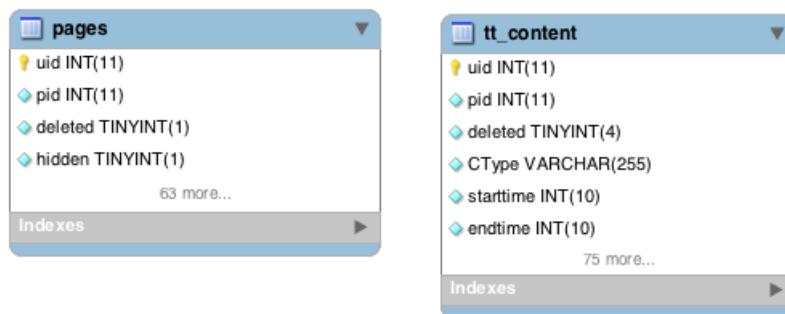


Abbildung 2.15: Die Tabellen *pages* und *tt_content*

Die Verknüpfung von Inhaltselement zu übergeordneter Seite erfolgt in der Datenbank über die Spalte *pid* (PageID), in der die ID der übergeordneten Seite als Fremdschlüssel gespeichert wird. Listing 19 zeigt die SQL-Abfrage, die alle Unterseiten einer Seite zurückgibt. Dabei hat die Seite, dessen Unterseiten abgefragt werden, die *uid* 4, die in die Abfrage eingesetzt wird. Die Anfrage lautet: Wähle alle Datensätze aus der Tabelle *pages*, die in der Spalte *pid* eine 4 stehen haben.

```
1 SELECT * FROM pages WHERE pid=4 ORDER BY sorting
```

Listing 19: Abrufen von Unterseiten einer Seite

Analog zum vorigen Listing, zeigt das Folgende die SQL-Anfrage, die alle Inhaltselemente von der Datenbank abfragt.

```
1 SELECT * FROM tt_content WHERE pid=4 ORDER BY sorting
```

Listing 20: Abrufen von Inhaltselementen einer Seite

Die beiden Abfragen geben alle Datensätze zurück, was in der Realität jedoch meistens nicht gewünscht ist. Zum Beispiel sollen keine gelöschten Datensätze angezeigt oder nicht alle Inhaltselemente ausgegeben werden. Datensätze werden in der Datenbank nicht gelöscht, sondern in der Spalte *deleted* durch das Setzen des Wertes auf 1 als gelöscht markiert. Inhaltselemente werden über die Spalte *CType* nach ihrem Typ gefiltert. Um das gewünschte Ergebnis zu erhalten müssen *WHERE*-Klauseln formuliert werden, was die doch recht trivialen SQL-Anfragen schnell komplex werden lässt.

In TYPO3 CMS können die Benutzerrechte sehr granular eingestellt werden. Die Einstellungen können per Benutzer oder Benutzergruppe vorgenommen werden. Dabei kann ein Benutzer Mitglied keiner, einer oder mehrerer Benutzergruppen sein. Zudem

kann eine Benutzergruppe keinen, einen oder mehrere Benutzer enthalten. Dies stellt eine Many-to-Many-Relation dar.

In der Datenbank wird die Zugehörigkeit von Benutzer <-> Gruppe von den Tabellen `fe_users` und `fe_groups` beziehungsweise `be_users` und `be_groups` abgebildet.

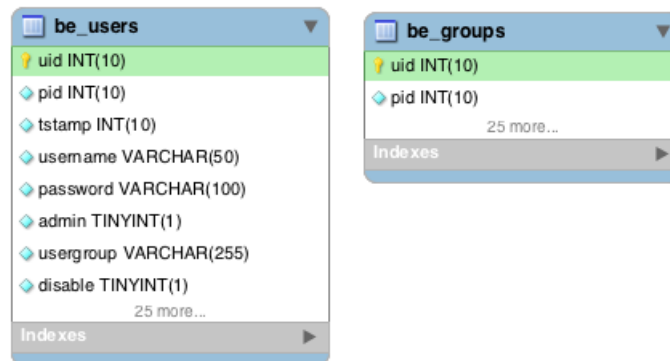


Abbildung 2.16: Die Tabellen `be_users` und `be_groups`

Hier fällt der Datentyp der Spalte `usergroup` auf, die ID der Gruppe des Benutzers speichert. Hier ist ein `VARCHAR` definiert, obwohl die Spalte `uid` den Typ `INT` hat. Dies liegt darin, dass die Zuordnung der Benutzer zu Gruppe über eine kommaseparierte Liste erfolgt:

id	pid	tstamp	username	password	admin	usergroup	disable	hidden	deleted
1	0	1191353353	admin	secret	1		0	0	0
2	0	1281556682	sname	secret	1	43	0	0	0
3	0	1191353353	hagrid	secret	0	5,32,43	0	1	0

Diese stellt lediglich ein Beispiel dar. Kommaseparierte Listen gibt es an vielen Stellen in der Datenbank. Die API von TYPO3 CMS stellt Methoden bereit, die die Liste für die weitere Verarbeitung aufbereiten.

Dieses Konstrukt existiert wahrscheinlich seit Beginn des Systems und es ist zu vermuten, dass damit eine Many-to-Many-Tabelle vermieden werden sollte, was die Komplexität der SQL-Anfrage erhöht.

Um die kommaseparierten Listen aufzulösen, müsste eine weitere Tabelle eingeführt werden, dessen zwei Spalten jeweils auf die `id` der beiden zu verknüpfenden Tabellen referenzieren.

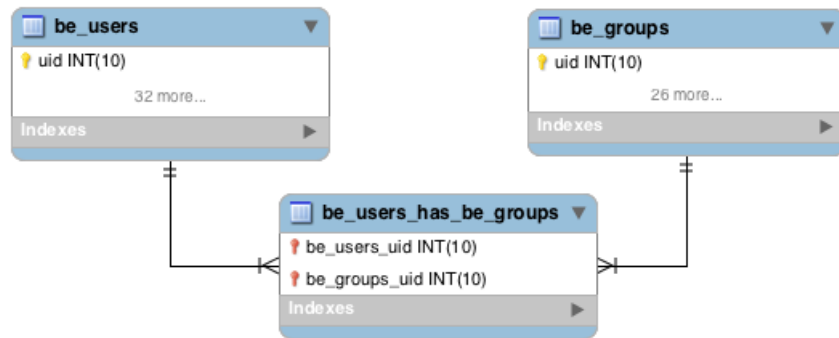
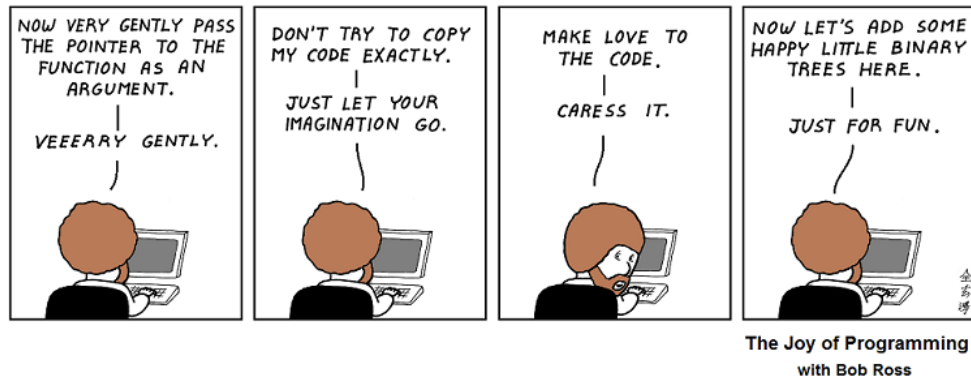


Abbildung 2.17: Normalisierung über Many-to-Many Tabelle

be_users_uid	be_groups_uid
2	34
3	5
3	32
3	43

TYPO3 CMS nutzt weder Datenbankseitige *Constraints* noch Fremdschlüssel Definitionen. Alle Referenzierungen werden von TYPO3 CMS selbst verwaltet.



3.1 Konzeption des Prototypen

Bevor mit der Implementierung des Prototyps begonnen werden konnte, wurden die zu erreichenden Ziele definiert.

- Der Prototyp erhält den Extensionkey *doctrine_dbal*.
- Der Prototyp ist eine normale Extension, die über das *Install Tool* installierbar ist. Dies ist notwendig, da bereits bei der Installation das zu nutzende DBMS auswählbar sein muß. Er ist gegenfalls ohne größeren Aufwand in eine Systemextension umwandelbar.
- Der Prototyp unterstützt die alte Datenbank API, damit TYPO3 CMS und externe Extensions weiterhin funktionieren.
- Der Prototyp unterstützt MySQL als DBMS.
- Die Methodennamen der neuen API folgen den TYPO3 Coding Guidelines (CGL).
- Die Erstellung der Basisdatenbank erfolgt durch Doctrine DBAL und nutzt dessen abstraktes Datenbankschema.
- Der Prototyp nutzt intern *Prepared Statements*.
- Der Prototyp führt eine *Fluent Query Language* ein, damit auf die manuelle Formulierung von SQL Anfragen verzichtet werden kann.

Diese Anforderungen konnten anschließend in einzelne Teilaufgaben zusammengefasst werden:

1. Erhöhung der Testabdeckung der vorhandenen Datenbank API
2. Erstellen der Grundstruktur des Prototypen

3. Integration in das Install Tool
4. Implementation einer Fluent API
5. Umbau von TYPO3 CMS auf die API des Prototypen

3.2 Vorbereitung

3.2.1 Installation von TYPO3 CMS

TYPO3 CMS wurde auf dem lokalen Rechner per *GIT* in der Entwicklerversion 6.2.x-dev unter [thesis.dev](#) installiert. TYPO3 CMS 6.2 stellte die zum Zeitpunkt der Implementierung aktuelle Version dar. Die Entwicklerversion wurde gewählt, um von der fortlaufenden Weiterentwicklung des Systems zu profitieren. Das Softwareversionierungsprogramm *GIT* wurde verwendet, damit der Code einfacher aktualisierbar war und eigene Änderungen daran nachvollzogen und dokumentiert werden konnten.

Abbildung 3.1 zeigt die Verzeichnisstruktur nachdem das System installiert und alle notwendigen Verzeichnisse und Symlinks erstellt wurden:

```
$ tree -L 2 --dirsfirst
.
├── http
│   ├── fileadmin
│   ├── typo3 -> typo3_src/typo3
│   ├── typo3_src -> ../typo3cms
│   ├── typo3conf
│   ├── uploads
│   └── index.php -> typo3_src/index.php
└── typo3cms
    ├── typo3
    ├── ChangeLog
    ├── GPL.txt
    ├── INSTALL.md
    ├── LICENSE.txt
    ├── NEWS.md
    ├── README.md
    ├── .htaccess
    ├── composer.json
    └── index.php
```

Abbildung 3.1: Die Grundstruktur von [thesis.dev](#)

Das Verzeichnis `http` wurde per Eintrag in der Apache2 Konfiguration als VirtualHost definiert.

```
<VirtualHost *:80>~
DocumentRoot "~/Sites/thesis.dev/http"~
ServerName thesis.dev~
</VirtualHost>
```

TYPO3 CMS wurde nach `typo3cms` installiert, damit dessen Dateien nicht über den Apache erreichbar sind.

Version: Commit: 66df0ce from 2014-05-22 15:52:28 +0200

Um die Installation unter der Adresse `thesis.dev` erreichen zu können, wurde in der Hostdatei ein A-Record angelegt.

```
sudo sh -c "echo '127.0.0.1 thesis.dev' >> /etc/hosts"
```

Im Anschluß daran wurde eine leere Datenbank mit dem Namen `thesis` erstellt:

```
mysql -u root -p
MariaDB [(none)]> create database if not exists thesis;
Query OK, 1 row affected (0.01 sec)
MariaDB [(none)]> quit;
```

Durch das Aufrufen von <http://thesis.dev/> im Browser wird der Installationsprozess gestartet, der in fünf Schritten das System installiert.

3.2.2 Schritt 1 - Systemcheck

Im ersten Schritt (Abb.: 3.2a) prüft das *Install Tool* ob alle Verzeichnisse und Symlinks angelegt wurden und die entsprechenden Benutzerrechte besitzen. Intern werden hier Verzeichnisse wie `typo3temp` und Dateien wie `LocalConfiguration.php` angelegt.

3.2.3 Schritt 2 - Eingabe der Datenbankdaten

Im zweiten Schritt (Abb.: 3.2b) werden die Benutzerdaten für die Datenbank eingegeben. Wenn anstelle von MySQL ein alternatives DBMS genutzt werden soll, kann über die Schaltfläche am Ende des Formulars die Systemextension DBAL installiert werden.

3.2.4 Schritt 3 - Auswahl der Datenbank

Nachdem die Verbindungsdaten eingegeben wurden, versucht TYPO3 CMS eine Verbindung zum DBMS zu etablieren. Gelingt dies, werden alle verfügbaren Datenbanken abgefragt und aufgelistet (Abb.: 3.2c). Hier kann über die Liste eine leere Datenbank ausgewählt oder über eine erstellende eingegeben werden.

3.2.5 Schritt 4 - Einrichten eines TYPO3 Administrators

In 4. Schritt (Abb.: 3.2d) der Installation wird ein Administrator eingerichtet und es kann ein Name für die Seite vergeben werden. Danach werden die Basistabellen in der Datenbank angelegt.

3.2.6 Schritt 5 - Abschluß der Installation

Mit diesem Schritt ist die Installation abgeschlossen.

Installing TYPO3 CMS 6.2.3-dev

1 2 3 4 5

System environment check

TYPO3 is an enterprise content management system that is powerful, yet easy to install.

After some simple steps you'll be ready to add content to your website. This first step checks your system environment and points out issues.

System looks good. Continue!

(a) Installation TYPO3 CMS - 1. Schritt

Installing TYPO3 CMS 6.2.3-dev

1 2 3 4 5

Database connection

If you have not already created a username and password to access the database, please do so now. This can be done using tools provided by your host.

Username

Password

Type

Host

Port

Socket

Continue

TYPO3 CMS native database implementation is based on MySQL. A database abstraction layer allows to run TYPO3 CMS on different database engines like postgres. This is used rather seldom and some core parts and extensions do not fully support this. Your TYPO3 CMS experience might suffer if you choose to install the system on anything different than MySQL.

I do not use MySQL

(b) Installation TYPO3 CMS - 2. Schritt

Installing TYPO3 CMS 6.2.3-dev

1 2 3 4 5

Select database

You have two options:

☒ Use an existing empty database:

-- Select database --

☐ OR create a new database:

Attention: The database user must have sufficient privileges to create the whole structure.

Enter a name for your TYPO3 database.

Continue

(c) Installation TYPO3 CMS - 3. Schritt

Installing TYPO3 CMS 6.2.3-dev

1 2 3 4 5

Create user and import base data

Import basic database structure and create a backend administrator user. The password can be used to log in to the **install tool** and the **TYPO3 CMS backend** (default username is "admin").

Username

Password

Show password ☐

Warning: This password gives an attacker full control over your instance if cracked. It should be strong (include lower and upper case characters, special characters and numbers) and must be at least eight characters long.

Site name

Continue

(d) Installation TYPO3 CMS - 4. Schritt

Installing TYPO3 CMS 6.2.3-dev

1 2 3 4 5

Installation done!

The only thing remaining is to set some configuration values based on your system environment, which happens automatically in this step. Then you will be redirected to your TYPO3 CMS backend, ready for you to log in with the user you just created.

Want a pre-configured site?

You now have an empty installation. If you want a pre-configured site, there are distributions on the web which can be installed via the Extension Manager. If you check the option below, the list of distributions will be fetched and you will be able to choose one directly. **Please note: This may take some time after login.**

☒ Yes, download the list of distributions.

Open the backend

(e) Installation TYPO3 CMS - 5. Schritt

Abbildung 3.2: Installation von TYPO3 CMS

3.2.7 Implementation von Unit Tests der alten Datenbank API

Um zu gewährleisten, dass TYPO3 CMS sowohl mit der alten API als auch mit der neuen API kompatibel ist, wurden Unit Tests für die alte Datenbank API geschrieben.

Die zur Ausführung der Unit Tests notwendige Extension *ext:phpunit* wurde über den *Extension Manager* installiert. Sie stellt das gleichnamige Testing Framework *PHPUnit*¹ zur Verfügung und bringt einen graphischen Testrunner mit.

Die alte Datenbank API verfügt zu diesem Zeitpunkt über 40 Tests mit 49 Assertions, welche jedoch lediglich Hilfsmethoden testen. Im Laufe der Arbeit wurden 68 Tests implementiert. Die Abbildung 3.3 zeigt die Ausführung der von TYPO3 CMS mitgelieferten Unit Tests; Abbildung 3.4 zeigt die Ausführung der neu implementierten sowie der alten Tests.

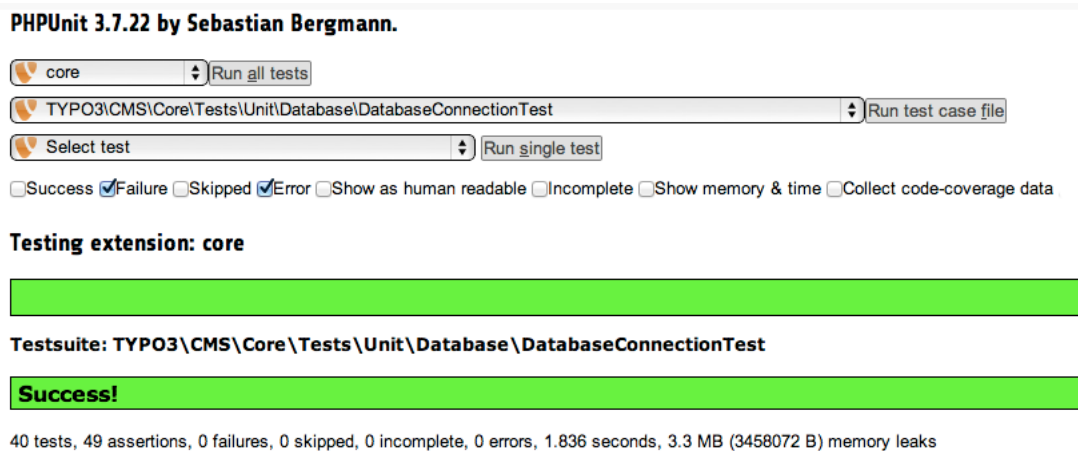


Abbildung 3.3: Ausführung der vorhandenen Unit Tests für die alte Datenbank API

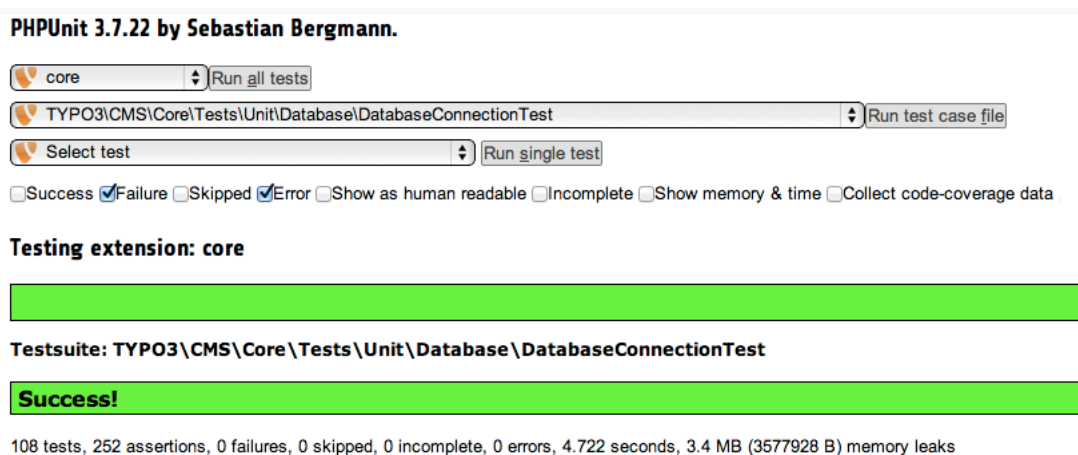


Abbildung 3.4: Ausführung der vorhandenen und hinzugefügten Unit Tests für die alte Datenbank API

¹ <http://www.phpunit.de>

3.3 Erstellung des Prototypen

Die Grundstruktur des Prototypen wurde unter `thesis/http/typo3conf/ext/doctrine_dbal` erstellt.

```

├── doctrine_dbal/
│   ├── Configuration/
│   ├── Resources/
│   ├── ext_emconf.php
│   ├── ext_icon.gif
│   └── ext_tables.php

```

Die Datei `ext_emconf.php` enthält die Metainformationen der Extension, die vom Extension Manager verarbeitet werden.

```

1  <?php
2  $EM_CONF[$_EXTKEY] = array(
3      'title' => 'Doctrine DBAL',
4      'description' => 'Doctrine DBAL Integration in TYPO3 CMS',
5      'category' => 'be',
6      'author' => 'Stefano Kowalke',
7      'author_email' => 'blueduck@gmx.net',
8      'author_company' => 'Skyfillers GmbH',
9      ...
10     'version' => '0.1.0',
11     'constraints' => array(
12         'depends' => array(
13             'typo3' => '6.2.0-6.2.99',
14         ),
15         'conflicts' => array('adodb', 'dbal'),
16     ),
17 );
18 );

```

Listing 21: Die Datei `ext_emconf.php`

Anschließend wurde Doctrine DBAL über *Composer* installiert, indem es als externe Abhängigkeit in der `composer.json` definiert und durch das Kommando `composer install` in den Ordner `vendor/doctrine` installiert wurde.

```

{
    "name": "typo3/doctrine_dbal",
    "type": "typo3-cms-extension",
    "description": "This brings Doctrine2 to TYPO3",
    "homepage": "http://typo3.org",
    "license": ["GPL-2.0+"],
    "version": "6.2.0",
    "require": {
        "doctrine/dbal": "dev-master"
    },
    "minimum-stability": "dev",
}

```

Listing 22: Die Datei `composer.json`

Die Integration von Doctrine DBAL sollte so transparent für TYPO3 CMS und die Extensions erfolgen, dass weiterhin über die Methoden der alten API auf die Datenbank zugegriffen werden kann. Die alte API steht dabei vergleichbar einer Fassade vor der neuen API, die ankommende Anfragen selbst behandelt oder an die neue API delegiert. Dieses Vorgehen erlaubt die sukzessive Integration von Doctrine DBAL.

Dazu wurde

- im Ordner `doctrine_dbal/Classes/Persistence/Doctrine` die Datei `DatabaseConnection.php` erstellt (im weiteren Verlauf als *neue API* bezeichnet),
- die Datei `DatabaseConnection.php` aus der Codebasis von TYPO3 CMS in den Ordner `doctrine_dbal/Classes/Persistence/Legacy` des Prototypen kopiert (im weiteren Verlauf als *alte API* bezeichnet),
- die Datei `DatabaseConnectionTests.php` aus der Codebasis von TYPO3 CMS in den Ordner `doctrine_dbal/Tests/Persistence/Legacy` des Prototypen kopiert,
- eine Vererbung realisiert, die die Klasse der alten API von der Klasse der neuen API erben lässt (siehe Abb.: 3.5 und

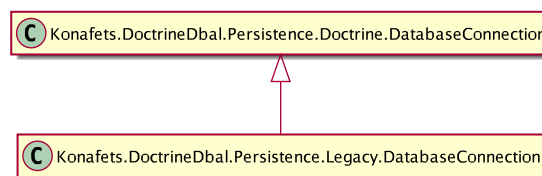


Abbildung 3.5: Alte API-Klasse erbt von neuer API-Klasse

- die Klasse der alten API per XCLASS in der Datei `ext_localconf.php` registriert (Listing 23).

```

1  if (!defined('TYPO3_MODE')) {
2      die('Access denied.');
```

```

3  }
4
5  $GLOBALS['TYPO3_CONF_VARS']['SYS']['Objects']
6  ['TYPO3\CMS\Core\Database\DatabaseConnection'] =
7  array(
8      'className' => 'Konafets\DoctrineDbal\Persistence\Legacy\DatabaseConnection');
```

Listing 23: Registrierung der XCLASSes in `doctrine_dbal/ext_localconf.php`

Danach wurden alle Eigenschaften aus der alten API in die neue API verschoben und mit Setter/Getter-Methoden versehen, die von der alten API ab diesem Zeitpunkt genutzt wurden.

Die Umstellung auf Doctrine begann mit dem Refactoring der Methode `connectDB()` der alten API. Dabei wurde lediglich die Implementation der Methode vereinfacht, da sie unübersichtlich war und mehrere unterschiedliche Aufgaben ausführte, die nichts mit deren Aufgabengebiet - der Herstellung einer Verbindung zur Datenbank - gemein hatten.

Neben ihrer eigentlichen Aufgabe hast sie

- einen Test durchgeführt, ob eine Datenbank konfiguriert ist
- die konfigurierte Datenbank ausgewählt und
- verschiedene Hooks ausgeführt

Code, der nicht zur definierten Aufgabe der Methode gehörte, wurde in eigene Methoden ausgelagert. Die Methode zur Überprüfung der veralteten Parameter konnte gänzlich entfallen, da die Änderungen an der neuen API stattfanden. Die Methode wurde anhand des aufgestellten Namensschemas umbenannt und wird von der alten Methode aufgerufen. Zusätzlich fängt die Methode nun die von Doctrine DBAL kommende Exception wenn keine Verbindung erstellt werden konnte und wirft eine eigene `ConnectionException`. Die entsprechenden Exceptionklassen wurden in `doctrine_dbal/Classes/Exceptions/` erstellt. Listing 24 zeigt die Methode nach dem Refactoring; Listing 25 zeigt den Aufruf der neuen Methode.

```

1  public function connectDatabase($isInitialInstallationInProgress = FALSE) {
2      // Early return if connected already
3      if ($this->isConnected) {
4          return;
5      }
6
7      if (!$isInitialInstallationInProgress) {
8          $this->checkDatabasePreconditions();
9      }
10
11     try {
12         $this->link = $this->getConnection();
13     } catch (\Exception $e) {
14         throw new ConnectionException($e->getMessage());
15     }
16
17     $this->isConnected = $this->checkConnectivity();
18
19     if (!$isInitialInstallationInProgress) {
20         if ($this->isConnected) {
21             $this->initCommandsAfterConnect();
22             $this->selectDatabase();
23         }
24
25         $this->prepareHooks();
26     }
27 }
```

Listing 24: connectDatabase() nach dem Refactoring

```

1  public function connectDB(...) {
2      ...
3      $this->connectDatabase();
4  }
```

Listing 25: connectDB() delegiert an die neue API-Methode

Die originale Implementation der Methode ist in `typo3/sysext/core/Classes/Database/DatabaseConnection.php` ab Zeile 1578 zu finden.²

Die Erstellung der Verbindung wurde in der alten API an die Methode `sql_pconnect()` ausgelagert. `getConnection()` ist der Name der Methode die diese Aufgabe in der neuen API übernimmt.

Zunächst wurde Code, welcher nicht zum Aufgabengebiet der Methode gehört, in eigene Klassen ausgelagert. In dem Fall betraf das den Test nach einer installierten MySQLi PHP-Erweiterung, welcher im gleichen Zuge auf PDO geändert wurde. Die Initialisierung von Doctrine erfolgt in einer eigenen Methode. Dort wird je eine Instanz von `\Doctrine\DBAL\Configuration` und `\Doctrine\DBAL\Schema\Schema` erstellt. In der alten API bot Die Methode die Möglichkeit einer persistenten Verbindung zur Datenbank. Dies wurde implementiert, in dem ein eigenes PDO-Objekt mit dem Konstrukturparameter `\PDO::ATTR_PERSISTENT => true` erstellt und in dem Konfigurationsarray gespeichert wurde. Dieses Konfigurationsarray (siehe Listing 26) wird `\Doctrine\DBAL\DriverManager::getConnection()` übergeben.

```

1  protected $connectionParams = array(
2      'dbname' => '',
3      'user'   => '',
4      'password' => '',
5      'host'   => 'localhost',
6      'driver' => 'pdo_mysql',
7      'port'   => 3306,
8      'charset' => 'utf8',
9  );

```

Listing 26: Das Konfigurationsarray für Doctrine DBAL

Zudem muß Doctrine DBAL der Datentyp `Enum` bekanntgemacht werden. Anschließend wird der Schema Manager erstellt, welcher zur Verwaltung des Datenbankschemas notwendig ist. Am Schluß wird das Verbindungsobjekt zurückgegeben. Der Code ist in `\Konafets\DoctrineDbal\Persistence\Doctrine\DatabaseConnect::getConnection()`³ zu finden.

Nachdem durch Doctrine DBAL die Verbindung hergestellt werden konnte, wurde die Methode `query()` der alten API übernommen. Sie stellt die zentrale Schnittstelle aller API-Methoden zum Senden ihrer Anfragen an die Datenbank dar. Sie kapselt die gleichnamige Methode `\Doctrine\DBAL\Connection::query()`.

```

1  protected function query($query) {
2      if (!$this->isConnected) {
3          $this->connectDatabase();
4      }
5
6      $stmt = $this->link->query($query);
7
8      return $stmt;
9  }

```

Listing 27: `getConnection()` der neuen API

² <http://bit.ly/typo3cms-legacy-connectdb>

³ <http://bit.ly/1s0XU1q>

Wie bereits in Kapitel 2.2.4 erwähnt wurde, gibt `query()` ein Statementobjekt zurück, welches die Ergebnismenge bereithält. Dort wurde ebenfalls dargestellt, dass diese Menge durch die Methode `fetch()` in Verbindung mit PDO-Konstanten in Index-basierte oder Assoziative Arrays formatiert werden kann.

Um die Arbeit mit der Ergebnismenge zu erleichtern wurden die drei Methoden `fetchAssoc($stmt)`, `fetchRow($stmt)` und `$fetchColumn($stmt)` implementiert, die die am Häufigsten vorkommenden *Fetch-Styles* kapseln.

```

1 public function fetchAssoc($stmt) {
2     if ($this->debugCheckRecordset($stmt)) {
3         return $stmt->fetch(\PDO::FETCH_ASSOC);
4     } else {
5         return FALSE;
6     }
7 }

```

Die ehemaligen `admin_*`-Methoden wurden in `list_*`-Methoden umbenannt, da die Unterscheidung in Admin und Nicht-Admin Methoden nicht nachvollziehbar war. Diese Methoden stellen wichtige Metainformation zur darunterliegenden Datenbank bereit, die nicht nur für das *Install Tool* von Nutzen sind.

Als Beispiel der Abstraktion, die Doctrine DBAL mitbringt, sei die Implementation der Methode `listDatabases()` angeführt. Nach dem obligatorischen Verbindungstest, gibt der `SchemaManager` eine Liste aller Datenbanken zurück - vollkommen unabhängig von der zugrundeliegenden DBMS.

```

1 public function listDatabases() {
2     ...
3     $databases = $this->schemaManager->listDatabases();
4     ...
5     return $databases;
6 }

```

Doctrine stellt zum Maskieren von Benutzereingaben die Methode `\Doctrine\DBAL\Connection::quote()` bereit. Im Gegensatz zur alten API, dessen Methode ein Backslash (\) den zu maskierenden Zeichen voranstellt, werden die Zeichen von Doctrine durch ein Hochkomma (') maskiert. Die Methode `\Doctrine\DBAL\Connection::quote()` wurde in der neuen API durch eine gleichnamige Methode gekapselt und stellt die Basis für alle weitere Methoden wie `fullQuoteString()`, `fullQuoteArray()`, `quoteString()` und `escapeStringForLike()` dar, welche das Verhalten der alten Methoden implementieren.

Die neue API bietet dagegen die Methoden `quoteColumn()`, `quoteTable()` und `quoteIdentifier()` zum Maskieren an. Es kann weiterhin `\Konafets\DoctrineDbal\Persistence\Doctrine\DatabaseConnection::quote()` genutzt werden; sicherer ist es jedoch von Anfang die Benutzung von Prepared Statements, welche von TYPO3 CMS seit Version 6.2 in Form eines Wrappers um die *Prepared Statements* von MySQLi angeboten werden. Sie besitzen die gleichen API, wie sie in Kapitel 2.2.4 vorgestellt wurde.

Durch den Aufruf der Methode `$stmt = $GLOBALS['TYPO3_DB']->prepare($sql)` wird zunächst ein Objekt vom Typ `\TYPO3\CMS\Core\Database\PreparedStatement` erstellt, dem der in `$sql` gespeicherte Prepared Statemnt übergeben wird. Ein Aufruf von `$stmt->bind(':lastName', 'Potter')` fügt einem internen Array des Objekts diese Werte hinzu. Zusätzlich wird versucht den Datentyp zu erkennen und ebenfalls zu speichern. Erst mit dem Aufruf von

`$stmt->execute()` wird die Datenbank kontaktiert. Zuvor wird jedoch erst die gespeicherte SQL-Anfrage und - die durch `bind()` - übergebenen Parameter von *Named Placeholder* in *Positional Parameter* transformiert, da MySQLi nur diese unterstützt. Daraufhin wird die SQL-Abfrage über `mysqli::prepare()` an die Datenbank gesendet. Danach werden die in dem Array gespeicherten Parameter per `mysqli::bind()` an die Abfrage gebunden und abschließend per `mysqli::execute()` ausgeführt. Zur Definition des Datentypes und des *Fetch-Styles* werden Konstanten wie `PreparedStatement::PARAM_INT` oder

`PreparedStatement::FETCH_ASSOC` verwendet.

Für die Umstellung auf Doctrine wurde die Datei

`typo3/sysext/core/Classes/Database/PreparedStatement.php` in das Verzeichnis `doctrine_dbal/Classes/Persistent` des Prototypen kopiert. Es wurden die eigens verwendeten Konstanten auf PDO-Konstanten umgemappt und die Transformation in *Positional Parameter* wurde entfernt, da von Doctrine beide Varianten unterstützt.

```

1  protected function guessValueType($value) {
2      if (is_bool($value)) {
3          $type = PDO::PARAM_BOOL;
4      } elseif (is_int($value)) {
5          $type = PDO::PARAM_INT;
6      } elseif (is_null($value)) {
7          $type = PDO::PARAM_NULL;
8      } else {
9          $type = PDO::PARAM_STR;
10     }
11
12     return $type;
13 }
```

3.4 Integration des Prototypen in das Install Tool

Um bereits bei der Installation ein alternatives DBMS nutzen zu können, mußte der Prototyp bereits über das *Install Tool* installierbar sein, was Anpassungen an dem *Install Tool* und TYPO3 CMS zur Folge hatte.

Wie in Kapitel 3.2.1 zu sehen war, besteht das *Install Tool* aus fünf Schritten, die durch die Klassen im Ordner `typo3/sysext/install/Classes/Controller/Action/Step` bereitgestellt werden. Der `\TYPO3\CMS\Install\Controller\StepController` iteriert bei jedem Reload des Installtools über alle Schritte und prüft ob der jeweils aktuelle Schritt bereits ausgeführt wurde oder noch ausgeführt werden muß. Er erkennt dies an Bedingungen, die von jedem Schritt definiert werden. Sind alle Bedingungen erfüllt, findet ein Redirekt auf den nächsten Schritt statt.

Die Ausgabe der Schritte erfolgt über verschiedene HTML-Template Dateien, die in der TYPO3 eigenen Template-Sprache *Fluid* verfasst sind. Das *Install Tool* setzt hier das MVC-Pattern ein, um die Geschäftslogik von der Präsentation zu trennen.

Die HTML-Templates unterteilen sich in *Layouts*, *Templates* und *Partials*, die in den jeweilig gleichnamigen Verzeichnissen in `typo3/sysext/install/Resources/Private/` zu finden sind.

- Ein Template beschreibt die grundlegende Struktur einer Seite. Typischerweise befindet sich darin der Seitenkopf und -fuß.

- Die Struktur einer einzelnen Seite wird von einem Template festgelegt.
- Partialen stellen wiederkehrende Elemente dar. Sie können in Layout- und Template-Dateien eingebunden werden. Die Schaltfläche *I do not use MySQL* aus Abbildung 3.2b in Kapitel 3.2.3 ist ein Beispiel für ein Partial.

Im folgenden werden die vorgenommenen Änderungen im Detail beschrieben.

- In `\TYPO3\CMS\Core\Core\Bootstrap` wurde die von Composer erstellte *Autoload*-Datei eingebunden. Somit können die von Doctrine DBAL zur Verfügung gestellten Klassen geladen werden. Siehe Listing 28
- Es wurden die Partialen `LoadDoctrineDbal.html` und `UnloadDoctrineDbal.html` zur De- und Installation des Prototypen sowie das Partial `DoctrineDbalDriverSelection.html` für die Auswahl des Datenbanktreibers erstellt
- diese wurden dem Template des zweiten Schritts `DatabaseConnect.html` hinzugefügt. Die Variable `isDoctrineEnabled` enthält den Installationsstatus des Prototypen. Abhängig von ihr wird entweder die Schaltfläche zur Installation des Prototypen oder das Auswahlfeld für die Datenbanktreiber und die Schaltfläche zur Deinstallation des Prototypen angezeigt.
- damit die Variable einen Wert enthält, wurde diese von der *Action* aus `\TYPO3\CMS\Install\Controller\Action\Step\DatabaseConnect` definiert und an die View übergeben.
- in `\TYPO3\CMS\Install\Controller\Action\Step\DatabaseConnect` wurden Methoden erstellt, die die installierten PDO-Extensions des Systems abfragen und an die View übergeben
- Zur Vermeidung leere Werte in der Konfigurationsdatei, wurden die Prüfungen der Benutzereingaben von `isset()` auf `!empty()` geändert.
- Das Eingabeformular für die Datenbankverbindungsinformationen wurde angepasst. Es wurde ein Feld für das *Charset* der Datenbank hinzugefügt und das Feld für die Datenbank entfernt, da sie in einem anderen Schritt über ein Auswahlfeld festgelegt wird.
- Damit die eingegebenen Daten weiterverarbeitet werden konnten, wurden diese in den entsprechenden PHP-Klassen ergänzt.
- Externe Abhängigkeiten werden vom Package Manager in `thesis.dev/http/Packages/Library` erwartet. Aus diesem Grund mußte der Ordner `doctrine_dbal/vendor/doctrine` nach `thesis.dev/http/Packages/Library/` kopiert werden.⁴

⁴ Composer Konfigurationsdateien werden seit TYPO3 CMS 6.2 analysiert. Aus den definierten Abhängigkeiten und den (System)-Extensions wird vom Package Manager ein Graph von Abhängigkeiten aufgebaut welcher in `thesis.dev/http/typo3conf/PackagesStates.php` gespeichert wird. Diese Datei wird bei der Installation erstellt und stetig aktualisiert. Da diese Funktionalität relativ neu ist, mußte diese Datei bei der Installation des Prototypen manuell angepasst werden.

```

1 // Bootloader.php
2 public function initializeClassLoader() {
3     /** Composer loader */
4     require_once PATH_typo3conf . 'ext/doctrine_dbal/vendor/autoload.php';
5
6     $classLoader = new ClassLoader($this->applicationContext);
7     ...
8 }

```

Listing 28: Einbinden des vom Composer erstellten Autoloaders

3.4.1 Doctrines Schemarepräsentation

Während der Installation werden die initialen Datenbanktabellen angelegt. Die dafür notwendigen SQL-Anweisungen halten die Extensions in *.sql-Dateien vor. Dabei handelt es sich um einfache Textdateien, die aus ein oder mehreren `CREATE TABLE` Anweisungen bestehen. Diese werden von `TYP03\CMS\Core\Database\SqlParser` geparkt, auseinandergenommen und neu zusammengesetzt, wobei kleinere Syntaxfehler behoben werden. Eine Hauptaufgabe des Parsers liegt darin, diejenigen *.sql-Dateien zu erkennen und zu vereinen, die die gleiche Tabelle mit unterschiedlichen Feldern anlegen wollen. Als Beispiel seien die Systemextensions `ext:frontend` und `ext:fellogin` erwähnt, die beide die Tabelle `fe_users` anlegen.

```

1 # ext:fellogin
2 CREATE TABLE fe_users (
3     fellogin_redirectPid tinytext,
4     fellogin_forgotHash varchar(80) default ''
5 );
6
7 # ext:frontend
8 CREATE TABLE fe_users (
9     uid int(11) unsigned NOT NULL auto_increment,
10    pid int(11) unsigned DEFAULT '0' NOT NULL,
11    tstamp int(11) unsigned DEFAULT '0' NOT NULL,
12    username varchar(50) DEFAULT '' NOT NULL,
13    password varchar(100) DEFAULT '' NOT NULL,
14    usergroup tinytext,
15    ...
16 );

```

Listing 29

Der Importvorgang erfolgt durch `\TYP03\CMS\Install\Controller\Action\Step\DatabaseData::importDatabaseData()`, die den Ist-Zustand der Datenbank mit dem Soll-Zustand vergleicht. Der Soll-Zustand wird durch den Service `\TYP03\CMS\Install\Service\SqlExpectedSchemaService::getTablesDefinitionString()` anhand der *.sql-Dateien ermittelt. Der Ist-Zustand wird durch `\TYP03\CMS\Install\Service\SqlSchemaMigrationService::getDatabaseExtra()` ermittelt und anschließend durch die Methoden `getDatabaseExtra()` und `getUpdateSuggestions()` derselben Klasse verglichen. Die genauere Analyse der Methoden würde für die Arbeit zu weit führen. Anschließend wird die Differenz in Form von SQL-Anweisungen an die Datenbank gesendet.

Statische Daten werden über Dateien mit der Bezeichnung `ext_tables_static+adt.sql` in die Datenbank eingefügt. Im Moment besitzt lediglich der Extension Manager solch eine Datei, die die URL zum TYPO3 Extension Repository (TER) einfügt.

Um die Abhängigkeit des *Install Tool* zu MySQL bei der Erstellung der Basistabellen aufzulösen wurden die *.sql-Dateien in die Schema Syntax von Doctrine überführt. Dies wurde bereits in Kapitel 2.2.2 durch die Erstellung eines Schemas skizziert. Die Zielstellung war die vollständige Umstellung auf die Nutzung von \Doctrine\DBAL\Schema\Schema-Objekten, da diese die notwendige Abstraktion von dem DBMS bieten und von Doctrine DBAL jederzeit anhand der aktuell verwendeten Plattform in die entsprechenden SQL-Anweisungen konvertiert werden können.

Dazu mußten die *.sql-Dateien der Systemextensions umgewandelt werden, sowie die Erstellung der dynamischen Cachetabellen implementiert werden. Im ersten Schritt wurden die *.sql-Dateien der folgenden Sytemextension in die Schemarepräsentation von Doctrine migriert:

- typo3/sysext/core
- typo3/sysext/extbase
- typo3/sysext/extensionmanager
- typo3/sysext/felogin
- typo3/sysext/filemetadata
- typo3/sysext/frontend
- typo3/sysext/impexp
- typo3/sysext/indexed_search
- typo3/sysext/indexed_search_mysql
- typo3/sysext/linkvalidator
- typo3/sysext/openid
- typo3/sysext/rsauth
- typo3/sysext/rtehtmlarea
- typo3/sysext/scheduler
- typo3/sysext/sys_action
- typo3/sysext/sys_note
- typo3/sysext/version
- typo3/sysext/workspaces

Die Datei ext_tables_static+adt.sql wurde in die Klasse DefaultData migriert.

Zur Ermittlung des Soll-Zustand der Datenbank wurde der Klasse \TYPO3\CMS\Install\Service\SqlExpectedSchemaService die Methode getTablesDefinitionAsDoctrineSchemaObject hinzugefügt, die die vorhandenen Schema.php-Dateien sucht und per **require** dem PHP-Skript zur Verfügung stellt. [Zeile 136] Die so eingebundenen Dateien werden zur weiteren Verarbeitung in einem Array gespeichert und als Parameter einer Funktion übergeben, die über die *Signal/Slot*-Implementation von TYPO3 CMS ein Signal emittiert. Die empfangende Methode erstellt daraufhin die Tabellen für das *Caching Framework*.

Diese werden von TYPO3 CMS dynamisch nach dem Muster `cf_cache_<tabellenname>` beziehungsweise `cf_cache_<tabellenname>_tags` erstellt. Die dazu implementierten Klassen `\TYPO3\CMS\Core\Cache\Schema\Typo3DatabaseBackendCacheSchema` und `\TYPO3\CMS\Core\Cache\Schema\Typo3DatabaseBackendTagsSchema` dienen dabei als Templates, denen bei der Instantiierung der `<tabellenname>` mitgegeben wird. Sie erstellen intern ein Objekt vom Typ `\Doctrine\DBAL\Schema\Schema`, welches schließlich die entsprechende Cachetabelle repräsentiert.

Einen Sonderfall stellen die Cache- und Tagstabellen für Extbase dar, da sie von TYPO3 CMS aus internen Gründen zunächst mit einem temporären Namen erstellt werden und erst im weiteren Verlauf in die Tabellen `cf_extbase_object` und `cf_extbase_object_tags` umbenannt werden können. Bereits hier zeigten sich die Vorteile durch die interne Verwendung von `\Doctrine\DBAL\Schema\Schema`-Objekten. Die Umbenennung konnte durch `renameTable()` des Objekts realisiert werden, anstelle der Nutzung von `str_replace()`, wie dies im originalen Code implementiert wurde.

Das von dem Signal zurückgegebene Schema-Array enthält nun alle zu erstellenden Tabellen - auch jene, die - wie oben beschrieben - von den Extensions mehrfach mit unterschiedlichen Feldern definiert wurden. Um diese Tabellen zu vereinen, wurde die Methode `flattenSchemas()` in der gleichen Klasse implementiert, der das Schema-Array übergeben wird. Das daraus resultierende Array stellt den Soll-Zustand der Datenbank dar.

Zur Ermittlung des Ist-Zustandes wurde die Klasse

`\TYPO3\CMS\Install\Service\SqlSchemaMigrationService` in die Extension kopiert und per XCLASS registriert. Sie wird von der Klasse `\Konafets\DoctrineDbal\Install\Service\SqlSchemaMigrationService` um die Methode `getCurrentSchemaFromDatabase()` erweitert, die den *Schema Manager* von Doctrine nutzt um den Zustand der Datenbank unabhängig vom DBMS abzufragen. Der gleichen Klasse wurde die Methode `getDifferenceBetweenDatabaseAndExpectedSchemaAsSql()` hinzugefügt um die Differenz zwischen dem Ist- und Sollzustand zu ermitteln. Sie gibt die Differenz in Form der zu erstellenden Tabellen in der jeweiligen SQL-Syntax des benutzten DBMS zurück.

Die kompletten Änderungen können unter

<http://bit.ly/typo3cms-integrate-schema-into-install-tool> für TYPO3 CMS und unter <http://bit.ly/prototype-integrate-schema-into-install-tool> nachvollzogen werden.

3.4.2 Installation des Prototypen

Schritt 1 - Systemcheck

Der erste Schritt entspricht dem aus Kapitel 3.2.2

Schritt 2 - Eingabe der Datenbankdaten

Im zweiten Schritt (Abb.: 3.6a) wird nun durch Betätigung der Schaltfläche *I want use Doctrine DBAL* Doctrine DBAL installiert, worauf eine entsprechende Meldung darüber zusammen mit dem Auswahlfeld für den Datenbanktreiber erscheint. Nach der Auswahl des Treibers werden die Inputfelder eingeblendet. Da die verschiedenen DBMS unterschiedliche Daten für den Aufbau einer Verbindung zur Datenbank benötigen, ist die Anzahl und Art der Felder von dem ausgewählten Treiber abhängig. Der Codeteil

ist in der Datei `typo3/sysext/install/Classes/Controller/Action/Step/DatabaseConnect.php` ab Zeile 556 zu finden. Abbildung 3.6b zeigt die Felder für MySQL.

Schritt 3 - Auswahl der Datenbank

Nachdem die Verbindungsdaten eingegeben wurden, versucht TYPO3 CMS eine Verbindung zum DBMS zu etablieren. Gelingt dies, werden alle verfügbaren Datenbanken abgefragt und aufgelistet (Abb.: 3.2c). Über die Auswahl kann eine leere Datenbank festgelegt werden. Alternativ kann über das Inputfeld eine zu erstellende Datenbank angegeben werden. Mit dem Absenden des Formulars werden die Basistabellen in der Datenbank angelegt.

Schritt 4 - Einrichten eines TYPO3 Administrators

Der Schritt entspricht dem aus Kapitel 3.2.5

Schritt 5 - Abschluß der Installation

Der Schritt entspricht dem aus Kapitel 3.2.6

The image contains two side-by-side screenshots of the TYPO3 CMS 6.2.3-dev installation wizard, specifically the 'Database connection' step. Both screenshots show a progress bar at the top with five steps, where step 2 is highlighted. A yellow banner at the top of both indicates 'Loaded Doctrine DBAL'.

Screenshot (a) - Schritt 2a: The form includes fields for 'Username', 'Password', 'Type' (set to 'Socket based connection'), 'Socket' (set to 'Default socket or enter name'), and 'Charset' (set to 'utf8'). Below these is a 'Continue' button and a checkbox labeled 'I want use Doctrine DBAL'.

Screenshot (b) - Schritt 2b: This form is similar to (a) but includes an additional 'Driver' dropdown menu at the top, which is set to 'MySQL PDO driver'. It also has a 'Database' field below the 'Socket' field. The 'Continue' button and the 'I want use Doctrine DBAL' checkbox are also present.

(a) Installation TYPO3 CMS mit Prototyp - Schritt 2a (b) Installation TYPO3 CMS mit Prototyp - Schritt 2b

Abbildung 3.6: Installation von TYPO3 CMS mit den Prototyp

Zur Überprüfung ob TYPO3 CMS tatsächlich den Prototypen nutzt, wurde in der IDE ein Debug-Breakpoint in `Konafets\DoctrineDbal\Persistence\Legacy\DatabaseConnection` innerhalb der `connectDB`

gesetzt, an dem die IDE die Ausführung von TYPO3 CMS anhielt als er erreicht wurde.

3.5 Umstellen der Query-Methoden auf Doctrine DBAL und Prepared Statements

In Kapitel 2.4 zur Datenbank API wurde bereits die Unterteilung in Generierende und Ausführende SQL-Methoden erläutert. Die Generierung der Anfrage erfolgt anhand der Parameter, die – zusammen mit den entsprechenden SQL-Anweisungen – in eine Zeichenkette umgewandelt und zurückgegeben werden.

```

1  public function UPDATEQuery($table, $where, array $fields_values, $no_quote_fields = FALSE) {
2      ...
3      $fields = array();
4      if (is_array($fields_values) && count($fields_values)) {
5          // Quote and escape values
6          $nArr = $this->fullQuoteArray($fields_values, $table, $no_quote_fields, TRUE);
7          foreach ($nArr as $k => $v) {
8              $fields[] = $k . '=' . $v;
9          }
10     }
11
12     $query = 'UPDATE ' . $table . ' SET ' . implode(',', $fields) .
13         ((string)$where !== '' ? ' WHERE ' . $where : '');
14     ...
15     return $query;
16 }
17 }
```

Listing 30: Original UPDATEQuery der alten Datenbank API

Doctrine DBAL bietet mit dem `QueryBuilder` eine Klasse an, mit der die Formulierung von SQL-Anweisungen abstrahiert werden kann. Die Generierung der SQL-Anweisung aus Listing 30 konnte durch die Nutzung des `QueryBuilder`s wie folgt verändert werden:

```

1  public function UPDATEQuery(...) {
2      ...
3      $query = $this->link->createQueryBuilder()
4          ->update('pages')
5          ->set('title', 'Foo')
6          ->where('id = 1')
7          ->getSQL();
8      ...
9  }
```

Listing 31: Generierung der SQL-Anfrage wird an den QueryBuilder delegiert

Dies wurde für alle generierenden Methoden der alten API realisiert. Dabei wurde der Code der Methoden in die neue API kopiert, während die alten Klassen den Aufruf an ihre Elternmethode delegieren `return parent::insertQuery(...)`. Dies wurde notwendig, da der Methodenname der alten API nicht der CGL entsprach; der Aufruf über `parent::` und nicht über `$this->` wurde notwendig, da PHP nicht zwischen Groß- und

Kleinschreibung unterscheidet - die Methoden `INSERTquery()` und `insertQuery()` werden als identisch angesehen.

Durch diesen Schritt wurde die letzte Abhängigkeit zu einem bestimmten DBMS aufgelöst.

Während die alte Datenbank API wie gewohnt benutzt werden kann (siehe Listing 32), wird für die neue Datenbank API eine Abstraktion zur Formulierung von SQL-Anweisungen nach dem Vorbild des `QueryBuilder`s angestrebt (siehe Listing 33). Wie zu sehen ist, werden dort auch die logischen Ausdrücke abstrahiert.

```

1  protected function resetStageOfElements($stageId) {
2      $fields = array('t3ver_stage' => \TYPO3\CMS\Workspaces\Service\StagesService::STAGE_EDIT_ID);
3      foreach ($this->getTcaTables() as $tcaTable) {
4          if (BackendUtility::isTableWorkspaceEnabled($tcaTable)) {
5              $where = 't3ver_stage = ' . (int)$stageId;
6              $where .= ' AND t3ver_wsid > 0 AND pid=-1';
7              $GLOBALS['TYPO3_DB']->exec_UPDATEquery($tcaTable, $where, $fields);
8          }
9      }
10 }
```

Listing 32

```

1  protected function resetStageOfElements($stageId) {
2      ...
3      $dbh = $GLOBALS['TYPO3_DB'];
4      $expr = $dbh->expr();
5      $query = $dbh->createUpdateQuery();
6
7      $query->update($tcaTable)
8          ->set('t3ver_stage', \TYPO3\CMS\Workspaces\Service\StagesService::STAGE_EDIT_ID)
9          ->where(
10             $expr->equals(t3ver_stage, $stageId),
11             $expr->greaterThan(t3ver_wsid, 0),
12             $expr->equals(pid, -1)
13         )
14     );
15
16     $query->execute();
17     ...
18 }
```

Listing 33: Ausblick auf die fertige Query-API

Es wurde darauf verzichtet die API des `QueryBuilder` direkt anzubieten. Die Gründe dafür sind

- vereinfachte API: ein `Query`-Objekt bietet lediglich die für seine Domain notwendigen Methoden an. Dadurch kann es auch nicht zur Formulierung von syntaktisch falschen Anfragen kommen.
- keine Abhängigkeit zu Doctrine DBAL: die Implementation der Methoden kann – wie am Anfang des Kapitels gezeigt - jederzeit gegen etwas anderes ausgetauscht werden.

Zur Umsetzung wurde das *Facade*-Entwurfsmuster angewendet. Hier stellt ein Domain-spezifisches query-Objekt die *Facade* dar und bietet nach außen lediglich eine Teilmenge der von *QueryBuilder* zur Verfügung gestellten Methoden.

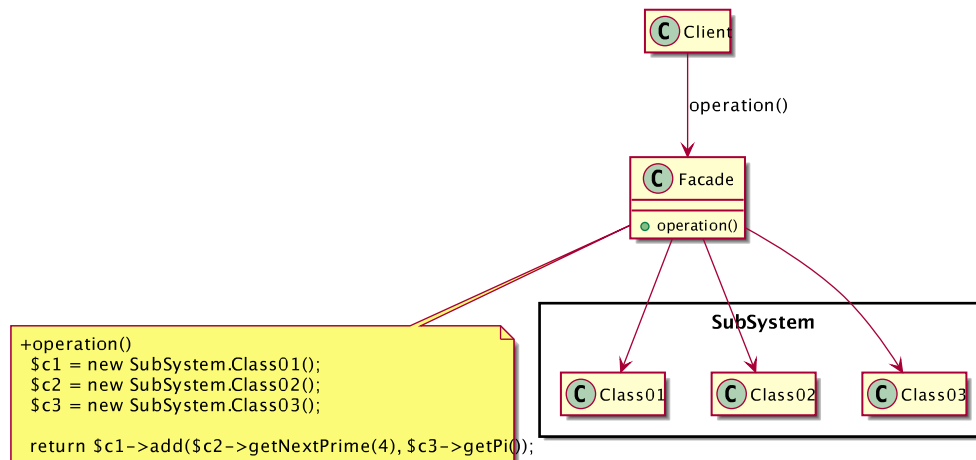


Abbildung 3.7: Schematischer Aufbau des Fassade-Entwurfsmuster

Das folgende UML-Diagramm zeigt anhand der *TruncateQuery*- und *UpdateQuery*-Objekte die fertige Hierarchie. Die Wurzel stellt das *queryInterface* dar, in dem alle Methoden festgelegt worden sind, die von allen query-Objekten implementiert werden müssen. Dadurch wird sichergestellt, dass jedes query-Objekt in Verbindung mit Prepared-Statements verwendet werden kann und die Kenntnis darüber besitzt wie es in eine SQL-Anweisung konvertiert und ausgeführt werden kann.

Das Interface wird von einem spezialisierten Interface erweitert, welche die Domain-spezifischen Methoden festlegt. So muß ein *TruncateQuery* die Methoden *truncate()*, *getType()* und *getSql()* implementieren. Die anderen vorgeschriebenen Methoden werden von der Klasse *phpinlineAbstractQuery* implementiert, die von jeder query-Klasse erweitert wird.

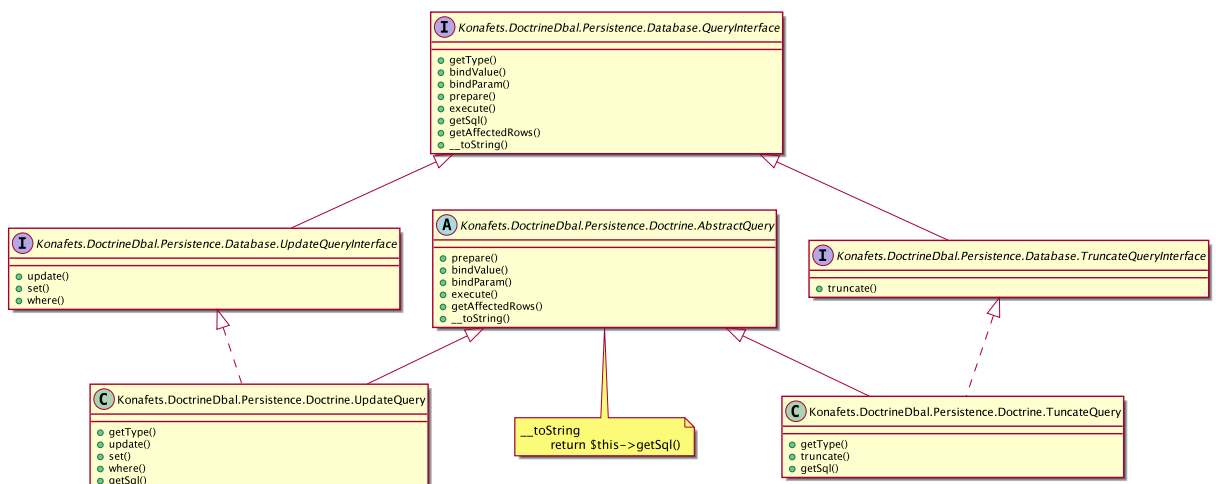


Abbildung 3.8: Aufbau der Query-API

Der Aufbau ist an die Implementation der Datenbank API von EzPublish angelehnt, die von Benjamin Eberlei eingeführt wurde.⁵ Während die Erzeugung der SQL-Anweisungen

⁵ <http://bit.ly/ezpublish-database-api>

innerhalb der EzPublish Datenbank API manuell erfolgt, delegieren die Methoden der Query-Klassen des Prototypen den Aufruf an den QueryBuilder.

```

1  public function update($table) {
2      $this->queryBuilder->update($table);
3
4      return $this;
5  }
6
7  public function set($columns, $values) {
8      $this->queryBuilder->set($columns, $values);
9
10     return $this;
11 }
12
13 public function where() {
14     $constraints = func_get_args();
15     $where = array();
16
17     foreach ($constraints as $constraint) {
18         if ($constraint !== '') {
19             $where[] = $constraint;
20         }
21     }
22
23     if (count($where)) {
24         call_user_func_array(array($this->queryBuilder, 'where'), $where);
25     }
26
27     return $this;
28 }
29
30 public function getSql() {
31     return $this->queryBuilder->getSQL();
32 }

```

Listing 34: Die Konvertierung des UpdateQuery erfolgt über den QueryBuilder

Der Klasse \Konafets\DoctrineDbal\Persistence\Doctrine\DatabaseConnection wurden create*Query()-Methoden hinzugefügt, die die Instantiierung eines Query-Objekts vereinfachen.

```

1  public function createUpdateQuery() {
2      if (!$this->isConnected) {
3          $this->connectDatabase();
4      }
5
6      return GeneralUtility::makeInstance(
7          '\\Konafets\\DoctrineDbal\\Persistence\\Doctrine\\UpdateQuery',
8          $this->link
9      );
10 }

```

Listing 35: Die Erzeugung eines UpdateQuery-Objekts

Nun konnte die Abhängigkeit zu Doctrine DBAL durch den direkten Aufruf des QueryBuilders aus Listing 31 entfernt werden:

```

1 public function UpdateQuery(...) {
2     ...
3     $query = $this->createUpdateQuery()
4         ->update('pages')
5         ->set('title', 'Foo')
6         ->where('id = 1')
7         ->getSQL();
8     ...
9 }

```

Listing 36

Zur Abstraktion der logischen Ausdrücke aus dem Codebeispiel 33 wurde die Klasse Expression implementiert, die auch als Fassade vom der von Doctrine DBAL zur Verfügung gestellten ExpressionBuilder. Das Namensschema folgt der Extbase-API.

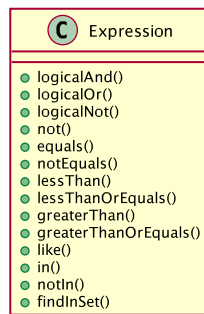


Abbildung 3.9: Die Klasse Expression

Die Query-API kann nun, wie es in Listing 33 gezeigt, verwendet werden.

Anhand eines weiteren Beispiels aus dem Code von TYPO3 CMS soll die Migration der alten API auf die neue API unter Verwendung des `UpdateQuery` gezeigt werden. Dazu wurde Code ausgewählt, welcher bei jedem Aufruf des Backends ausgeführt wird und somit gut testbar ist. Er befindet sich in der Datei

`typo3/sysex/core/Classes/Authentication/AbstractUserAuthentication.php` in der Methode `fetchUserSession()`. Der Code aktualisiert die Session für den angemeldeten Backendbenutzer. Die erzeugte SQL-Anweisung sieht so aus:

`UPDATE be_sessions SET ses_tstamp = '1400788819' WHERE ses_id='c2c75...' AND ses_name='be_typo_user'.`
 Die Werte in der `WHERE`-Bedingung werden durch `fullQuoteStr()` maskiert um SQL-Injections zu verhindern.

```

1 public function fetchUserSession($skipSessionUpdate = FALSE) {
2     ...
3     if ($timeout > 0 && $GLOBALS['EXEC_TIME'] < $user['ses_tstamp'] + $timeout) {
4         if (!$skipSessionUpdate) {
5
6             $this->db->exec_UPDATEQuery(
7                 $this->session_table,
8                 'ses_id=' . $this->db->fullQuoteStr($this->id, $this->session_table) .
9                 ' AND ses_name=' . $this->db->fullQuoteStr($this->name, $this->session_table),
10                array('ses_tstamp' => $GLOBALS['EXEC_TIME']));
11
12            // Make sure that the timestamp is also updated in the array
13            $user['ses_tstamp'] = $GLOBALS['EXEC_TIME']
14        }
15    } else {
16        // Delete any user set...
17        $this->logout();
18    }
19 }
20 }

```

Listing 37: fetchUserSession() im Original

Das folgende Listing zeigt den gleichen Code nach der Migration. Obwohl die Methoden bereits weiter oben vorgestellt wurden, soll hier auf die Verwendung eines `FluentInterface` hingewiesen werden. Dies ermöglicht zum einen die Verkettung der Methoden und zum anderen eine lesbare API, bei der auf den ersten Blick der Kontext eines jeden Parameters anhand des Methodennamens deutlich wird. Realisiert werden `FluentInterface` durch die Zurückgabe des eigenen Objekts in der Methode mittels `return $this`.

```

1     ...
2     if ($timeout > 0 && $GLOBALS['EXEC_TIME'] < $user['ses_tstamp'] + $timeout) {
3         if (!$skipSessionUpdate) {
4
5             $query = $this->db->createUpdateQuery();
6             $query->update($this->session_table)
7                 ->set('ses_tstamp', $GLOBALS['EXEC_TIME'])
8                 ->where(
9                     $query->expr->equals('ses_id', $this->db->quote($this->id)),
10                    $query->expr->equals('ses_name', $this->db->quote($this->name))
11                )->execute();
12        }
13    } else {
14        // Delete any user set...
15        $this->logout();
16    }
17    ...
18 }

```

Listing 38: fetchUserSession() nach der Migration mit manueller Maskierung

Wie zu sehen ist, wurde die Maskierung auch hier manuell durch `DatabaseConnection::quote()` vorgenommen. In Kapitel 2.2.4 wurde auf die Gefahren von SQL-Injections hingewie-

sen und wie diese durch *Prepared Statements* (siehe Kapitel 2.2.4) verhindert werden können. Das nächste Listing zeigt wie diese mit der API genutzt werden können.

```

1  $query = $this->db->createUpdateQuery();
2  $query->update($this->session_table)
3      ->set('ses_tstamp', $GLOBALS['EXEC_TIME'])
4      ->where(
5          $query->expr->equals('ses_id', $query->bindValue($this->id)),
6          $query->expr->equals('ses_name', $query->bindValue($this->name))
7      )->execute();
8  }
```

Listing 39: fetchUserSession() in Verbindung mit PreparedStatements

Die Methode `quote()` wurde gegen `bindValue(...)` ausgetauscht. Diese erzeugt einen *Named Parameter* und bindet den Wert der Variablen an das *PreparedStatement*-Objekt, dass sich in `$query` befindet. Analog dazu existiert die Methode `bindParam()`, die nach dem Vorbild der gleichnamigen Methode von Doctrine DBAL funktioniert. Die erzeugte SQL-Anfrage lautet:

```

UPDATE be_sessions SET ses_tstamp = 1400788396
WHERE (ses_id = :placeholder1) AND (ses_name = :placeholder2).
```

Somit konnte auch die Benutzung von *Prepared Statements* abstrahiert und vereinfacht werden. Es ist nun nicht mehr notwendig die Werte explizit an die SQL-Anweisung zu binden. Ein weiterer Schritt könnte eine grundsätzliche Erzeugung von *Prepared Statements* sein, von der ein Nutzer der Datenbank API nichts mitbekommt.

Für einfache SQL-Abfragen empfiehlt das Doctrine Projekt die Benutzung der Methoden `\Doctrine\DBAL\Connection\delete()`, `\Doctrine\DBAL\Connection\update()` und `\Doctrine\DBAL\Connection\insert()` die intern *Prepared Statements* mit *Positional Parametern* erzeugen. Zur Nutzung dieser Methoden wurden in der Klasse `\Konafets\DoctrineDbal\Persistence\Doctrine\DatabaseConnection` die Methoden gleichen Namens implementiert, die die Anfrage direkt an die Doctrine Methoden weiterreichen. Die neuimplementierten Methoden `executeDeleteQuery()`, `executeInsertQuery()`, `executeUpdateQuery()` verfolgen ebenfalls diesen Ansatz, wurden jedoch als Ersatz zu den `exec_*`-Methoden der alten API konzipiert. Diese konnten jedoch ohne tiefgreifende Eingriffe in den Code von TYPO3 CMS nicht verwendet werden, da der aufrufende Code von `exec_*`-Methoden ein `PDOStatement`-Objekt erwartet, welches für weitere Funktionen wie `$GLOBALS['TYPO3_DB']->sql_num_rows($res)` genutzt wird. Die Doctrine Methoden geben jedoch lediglich die Anzahl der veränderten Zeilen zurück - was im Grunde genau der Wert ist, der durch den Aufruf von `sql_num_rows()` erwartet wird.

QUELLENVERZEICHNIS

- [Ada95] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Englisch. Auflage: Reissue. Valley View, Calif.: Del Rey, Sep. 1995. isbn: 9780345391803.
- [Dam10] Karsten Dambekalns. *FrOSCamp 2010 in Zürich*. Englisch. Sep. 2010. url: <http://karsten.dambekalns.de/blog/froscamp-2010-in-zurich.html> (besucht am 08.04.2014).
- [DEM14] Karsten Dambekalns, Benjamin Eberlei und Christian Müller. *The reasons why TYPO3 Flow switched to Doctrine ORM*. microblog. Apr. 2014. url: <https://twitter.com/konafets/status/453102477081341952> (besucht am 08.04.2014).
- [DRB08] Dmitry Dulepov, Ingo Renner und Dhiraj Bellani. *TYPO3 extension development developer's guide to creating feature-rich extensions using the TYPO3 API*. English. Birmingham, U.K.: Packt Pub., 2008. isbn: 9781847192134 1847192130 1847192122 9781847192127.
- [Ebe12] Benjamin Eberlei. *Extension that replaces TYPO3 Extbase ORM with Doctrine2*. Englisch. Apr. 2012. url: <https://github.com/simplethings/typo3-extbase-doctrine2-extension> (besucht am 09.04.2014).
- [Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. en. Addison-Wesley Professional, 2003. isbn: 9780321127426.
- [ISO92] ISO/IEC. *Database Language SQL*. Juli 1992. url: <http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt> (besucht am 05.05.2014).
- [Lab+06] Kai Laborenz u. a. *TYPO3 4.0: das Handbuch für Entwickler ; [eigene Extensions programmieren ; TypoScript professionell einsetzen ; barrierefreie Websites, Internationalisierung und Performancesteigerung ; inkl. AJAX und TemplaVoila]*. German. Bonn: Galileo Press, 2006. isbn: 9783898428125 3898428125.
- [Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Englisch. Auflage: 1. Upper Saddle River, NJ: Prentice Hall, Aug. 2008. isbn: 9780132350884.
- [Mar12] Thomas Maroschik. *WIP: Doctrine DBAL integration*. Englisch. Dez. 2012. url: <https://git.typo3.org/Packages/TYP03.CMS.git/tree/b0a64733d2de2396c1f91dd559b37f0a4b652766> (besucht am 09.04.2014).
- [Pop07] Dennis Popel. *Learning PHP Data Objects: A Beginner's Guide to PHP Data Objects, Database Connection Abstraction Library for PHP 5*. Packt Publishing Ltd, 2007.
- [T3N09] T3N. Jonathan Wage über seinen Einstieg bei Doctrine und die Zukunft des Projekts: "Wir hatten gute Entwickler, aber keine Vision". Dez. 2009. url: <http://t3n.de/magazin/jonathan-wage-seinen-einstieg-doctrine-zukunft-projekts-224058/> (besucht am 01.05.2014).
- [TYP08] TYPO3 Association. *Berlin Manifesto*. Englisch. Okt. 2008. url: <http://typo3.org/roadmap/berlin-manifesto/> (besucht am 06.04.2014).
- [TYP13] TYPO3 Core Team Mailingliste. *TYPO3 - Team Core - RFC: Integrate Doctrine2 into TYPO3 CMS*. Englisch. Jan. 2013. url: <http://typo3.3.n7.nabble.com/RFC-Integrate-Doctrine2-into-TYP03-CMS-td237490.html> (besucht am 09.04.2014).

GLOSSAR

dbal

a very long description of of what is dbal

ABKÜRZUNGSVERZEICHNIS

API	Application Programming Interface
BE	Backend
CGL	Coding Guidelines
CMS	Content Management-System
DBAL	Database Abstraction Layer
DBMS	Database Management System
ECMS	Enterprise Content Management-Sytem
EM	Extension Manager
GPL2	GNU General Public License v.2
IDE	Integrated Development Environment
JCR	Content Repository for Java Technology API
MVC	Model-View-Controller
ORM	Object-relational mapping
PDO	PHP Data Objects
PHP	PHP: Hypertext Processor
SQL	Structured Query Language
T3Assoc	TYPO3 Association
TER	TYPO3 Extension Repository
WCMS	Web Content Management-Sytem

TABELLENVERZEICHNIS

2.1	Typkonvertierung von Doctrine nach MySQL und PostgreSQL	12
-----	---	----

ABBILDUNGSVERZEICHNIS

2.1	Zeitachse der TYPO3 CMS Entwicklung	4
2.2	Schematischer Aufbau von TYPO3	6
2.3	Schematischer Aufbau von Doctrine	9
2.4	Die beiden PDO-Klassen	14
2.5	Das Adapter Entwurfsmuster in UML-Notation	14
2.6	Aufbau des Connection-Objekts von Doctrine DBAL	15
2.7	Aufbau des Statement-Objekts von Doctrine DBAL	16
2.8	xkcd: Exploits of a mom ⁶	23
2.9	DatenbankConnection mit den generierenden Methoden	29
2.10	DatenbankConnection mit den ausführenden Methoden	30
2.11	DatenbankConnection: Methoden zur Verarbeitung der Ergebnismenge	30
2.12	DatenbankConnection: Hilfsmethoden	31
2.13	DatenbankConnection mit administrativen Methoden	31
2.14	Die Klasse PreparedStatement mit ausgewählten Methoden	32
2.15	Die Tabellen pages und tt_content	33
2.16	Die Tabellen be_users und be_groups	34
2.17	Normalisierung über Many-to-Many Tabelle	35
3.1	Die Grundstruktur von thesis.dev	37
3.2	Installation von TYPO3 CMS	39
3.3	Ausführung der vorhandenen Unit Tests für die alte Datenbank API	40
3.4	Ausführung der vorhandenen und hinzugefügten Unit Tests für die alte Datenbank API	40
3.5	Alte API-Klasse erbt von neuer API-Klasse	42
3.6	Installation von TYPO3 CMS mit den Prototyp	51
3.7	Schematischer Aufbau des Fassade-Entwurfsmuster	54
3.8	Aufbau der Query-API	54
3.9	Die Klasse Expression	56

LIST OF LISTINGS

1	Speichern eines Studenten in die Datenbank ohne ORM	10
2	Speichern eines Studenten in die Datenbank mit ORM	10
3	Erstellen eines Schemas mit Doctrine	11
4	Das erstellte Schema als MySQL Anfrage	12
5	Das erstellte Schema als PostgreSQL	12
6	16
7	17
8	17
9	18
10	20
11	21
12	22
13	24
14	25
15	25
16	Aktualisierung des Zeitpunkt des letzten Logins	29
17	Löschen eines Datensatzes aus einer Tabelle	29
18	Positional und Named Prepared Statements der TYPO3 CMS Datenbank API	32
19	Abrufen von Unterseiten einer Seite	33
20	Abrufen von Inhaltselementen einer Seite	33
21	Die Datei ext_emconf.php	41
22	Die Datei composer.json	41
23	Registrierung der XCLASSES in doctrine_dbal/ext_localconf.php	42
24	connectDatabase() nach dem Refactoring	43
25	connectDB() delegiert an die neue API-Methode	43
26	Das Konfigurationsarray für Doctrine DBAL	44
27	getConnection() der neuen API	44
28	Einbinden des vom Composer erstellten Autoloaders	48
29	48
30	Original UPDATEQuery der alten Datenbank API	52
31	Generierung der SQL-Anfrage wird an den QueryBuilder delegiert . . .	52
32	53
33	Ausblick auf die fertige Query-API	53
34	Die Konvertierung des UpdateQuery erfolgt über den QueryBuilder . .	55
35	Die Erzeugung eines UpdateQuery-Objekts	55
36	56
37	fetchUserSession() im Original	57
38	fetchUserSession() nach der Migration mit manueller Maskierung . . .	57
39	fetchUserSession() in Verbindung mit PreparedStatements	58

EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Flensburg, den 23. Mai 2014

Stefan Kowalke