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 <blueduck@gmx.net>
Matrikelnummer:	485366
Abgabedatum:	26. Mai 2014
Erstprüfer:	Prof. Dr. Hans-Werner Lang
Zweitprüfer:	Dipl. VK Tobias Hiep

Dieses Dokument wurde am 26. Mai 2014 mit LaTeX 2ε gesetzt.

Schrift:	Palatino 10pt
Typographie:	2012/07/29 v3.11b KOMA-Script
System:	LuaTeX-0.76.0 auf OSX 10.8
Editor:	Mou 0.8.5 beta und VIM 7.4

ABSTRACT

Web applications are often developed around one particular database management system. In the past MySQL was used a lot. To decouple the web application from this dependency you could use the database abstraction layer PDO, which is bundled with PHP. Doctrine DBAL goes one step further which offers, based on PDO, a unified interface to a broad range of other database management systems. Additionally Doctrine DBAL is the base of Doctrine ORM – a framework for object-relational database mapping, which is used by TYPO3 Flow and TYPO3 Neos. This paper demonstrates how to decouple the dependency to MySQL by integrating Doctrine DBAL into the content manangement system TYPO3 CMS. On top of that it implements an abstract query language for the underlying database with the choice to use Prepared Statements.

ZUSAMMENFASSUNG

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 objektrelationalen 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-Manangement-System TYPO3 CMS die Abhängigkeit zu MySQL entfernt werden kann. Zusätzlich wird eine abstrakte Abfragesprache für die zugrundeliegende Datenbank eingeführt, die eine wahlweise Nutzung von Prepared Statements anbietet.

"'The Babel fish,' said The Hitchhiker's Guide to the Galaxy quietly, 'is small, yellow and leech-like, and probably the oddest thing in the Universe.

...

The practical upshot of all this is that if you stick a Babel fish in your ear you can instantly understand anything in any form of language."'

– Douglas Adams, The Hitchhiker's Guide to the Galaxy

1

EINLEITUNG

Das Content Management System TYPO3 CMS nutzt standardmäßig die Datenbank MySQL. Wird stattdessen ein anderes Datenbank Management System wie Postgres, Oracle oder MSSQL verwendet, muss die optionale TYPO3 CMS Systemextension *DBAL* installiert werden. Diese verwendet zur Konvertierung der SQL-Abfragen in die jeweiligen SQL-Dialekte die externe Bibliothek *ADOdb* als Datenbankabstraktionsschicht.

Während DBAL stets an neue Versionen von TYPO3 CMS angepasst wurde, schien die Entwicklung von ADOdb im Jahr 2012 in einen Dornröschenschlaf verfallen zu sein[1], was innerhalb der TYPO3 Entwicklergemeinschaft die Frage nach einem Nachfolger der Abstraktionsschicht aufwarf. Als Kanditaten kamen die Projekte PDO[2], Propel[3] und Doctrine[4] in Frage. Am Ende konnte Doctrine überzeugen, da es unter aktiver Entwicklung steht und als einziges Projekt die Datenbankabstraktionsschicht von der Komponente der objektrealationale Abbildung voneinander getrennt anbietet. Nicht zuletzt waren die weite Verbreitung und die guten Erfahrungen, die die Schwesterprojekte von TYPO3 CMS, TYPO3 Flow und TYPO3 Neos, sammelen konnten ausschlaggebend.

Aber wie sieht es mit der Integration von Doctrine DBAL in TYPO3 CMS aus? Kann die Integration unter Beibehaltung der Kompatibilität zur existierenden Datenbank API realisiert werden? Und ist es möglich die Nutzung von Prepared Statements so weit zu vereinfachen, dass sie bevorzugt benutzt werden?

Um diese Fragen zu beantworten, soll im Rahmen der Bachelor-Thesis ein Prototyp erstellt werden, der Doctrine DBAL in TYPO3 CMS integriert und zudem eine abstrakte Abfragesprache implementiert, die eine wahlweise Benutzung von Prepared Statements ermöglicht. Als Ergebnis wird die komplette Auflösung der Abhängigkeit von TYPO3 CMS zu MySQL angestrebt. Ferner soll durch die Einführung einer abstrakten Abfragesprache die Benutzerschnittstelle mit der Datenbank vereinfacht und durch die Möglichkeit der Verwendung von Prepared Statements, die Sicherheit vor SQL-Injections erhöht werden.

Die Arbeit ist in vier Teile unterteilt:

Nach der Einführung folgt mit Kapitel zwei die Vermittlung der theoretischen Grundlagen. Es wird näher auf TYPO3 CMS, Doctrine DBAL, Prepared Statements und SQL-Injections eingegangen. Abgerundet wird das Kapitel mit der Beschreibung der Arbeitsweise und verwendeten Werkzeuge.

Im dritten Kapitel wird die Situation zu Beginn der Bachelor-Thesis beschrieben. Zur Sprache kommen die aktuelle Datenbank API, die TYPO3 CMS eigene Implementation der Prepared Statements, sowie das verwendete Datenbankschema.

Das vierte Kapitel widmet sich der Konzeption und Umsetzung des Prototypen. Es geht auf die Vorgehensweise zu dem anfänglichen Refactoring ein und beschreibt die Erstel-

1 Erst während der Arbeit an dieser Thesis konnte wieder Aktivität bei der Entwicklung von ADOdb verzeichnet werden. http://adodb.sourceforge.net/docs-adodb.htm#changelog
2 http://www.php.net/manual/de/book.pdo.php
3 http://propelorm.org/
4 http://doctrine-project.org

lung des Prototypen sowie dessen Integration in den Installationsprozess von TYPO3 CMS. Abgerundet wird das Kapitel mit der Implementation einer neuen Abfragesprache.

2

GRUNDLAGEN

2.1 *TYPO3 CMS*

TYPO3 CMS ist ein klassisches Web Content Management-System (WCMS), welches auf die Erstellung, die Bearbeitung und das Publizieren von Inhalten im Intra- oder Internet spezialisiert ist. Es 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. Mittlerweile wird es von einer freiwilligen Programmierergemeinde weiterentwickelt.

Laut der Website T3Census[1] gab es am 11. Mai 2014 301,637 Installationen von TYPO3 CMS.

2.1.1 *Architektur und Aufbau von TYPO3 CMS*

TYPO3 CMS wurde in der Programmiersprache PHP: Hypertext Processor (PHP) - basierend auf dem Konzept der Objektorientierung - geschrieben und ist damit auf jeder Plattform lauffähig, die über einen PHP Interpreter verfügt. PHP bildet zusammen mit einem Apache Webserver und einer MySQL Datenbank den sogenannten Webstack, der für alle gängigen Plattformen erhältlich ist.

Ansichtssache

Aus Anwendersicht teilt sich TYPO3 CMS in zwei Bereiche:

- das Backend
  stellt die Administrationsoberfläche dar. Hier erstellen und verändern Redakteure 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])

1 http://t3census.info/

Der Systemkern

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])

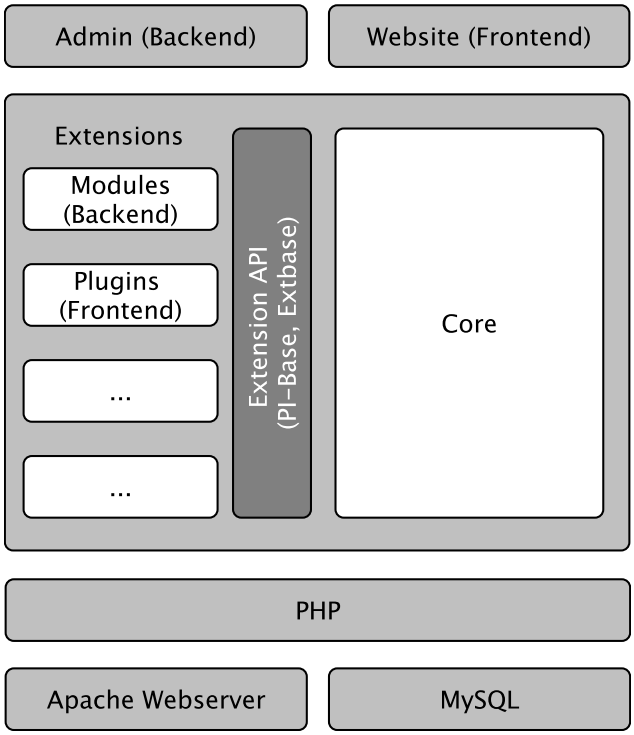


Abbildung 2.1: Schematischer Aufbau von TYPO3

Die Gesamtheit aller von TYPO3 CMS zur Verfügung gestellten Application Programming Interface (API)s, 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. 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, Löschen und Leeren[2] von Datensätzen bereitzustellen hat. Auf die aktuelle Datenbank-API wird in Kapitel 3 näher eingegangen.

XCLASS

TYPO3 CMS besitzt einen – als XCLASS bezeichneten – Mechanismus, der es erlaubt Klassen zu erweitern oder Methoden mit eigenem Code zu überschreiben. Dies funktioniert für den Systemkern wie auch für andere Extensions. Damit eine Klasse per

2 CRUD - **C**reate, **R**etrieve, **U**pdate und **D**elete

XCLASS erweiterbar ist, darf sie nicht per `new()` Operator erzeugt werden, sondern durch `\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 er 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 Systems instanziiert. Das kann dazu führen, dass Klassen die als Singleton ausgeführt sind, nicht überschrieben werden können oder es kann zu unvorhergesehenen Nebeneffekten kommen.

2.1.2 *Extensions*

Extensions sind funktionale Erweiterungen, die in System-, globale und lokale Extensions unterteilt werden. Sie interagieren mit dem Systemkern über die Extension API und stellen die Möglichkeit dar TYPO3 CMS zu erweitern und anzupassen.

Systemextensions 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.

Neben Systemextensions gibt es globale[3] und lokale Extensions. Lokale Extensions werden im Ordner `typo3conf/ext/` und globale Extensions im Ordner `typo3/ext` installiert.

Extension Manager

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

3 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.

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*[4] (vgl. [T3N09]).

Das Projekt wurde 2006 von Konsta Vesterinen initiiert[5] und im Jahr 2008 als Version 1.0.0 veröffentlicht.

Die beiden bekanntesten Produkte des Projekts sind:

- Object-relational mapping (ORM) - ermöglicht die objektrelationale Abbildung von Objekten auf Datenbanktabellen
- Database Abstraction Layer (DBAL) - stellt eine Datenbankabstraktionsschicht bereit

Wie aus Abbildung 2.2 ersichtlich wird, ist Doctrine DBAL lediglich als eine dünne Schicht auf von PHP Data Objects (PDO) ausgeführt, die die grundlegenden Funktionen zur Abstraktion von Datenbanken implementiert. Die ORM Schicht baut demnach auf Doctrine DBAL auf.

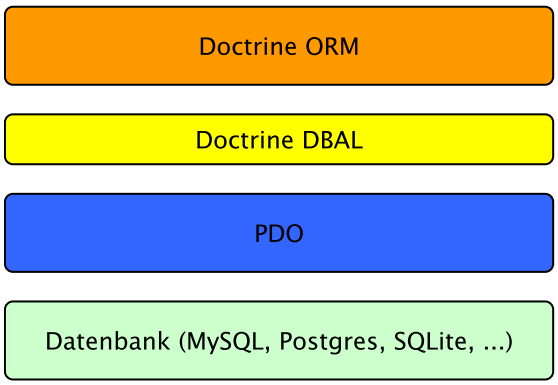


Abbildung 2.2: Schematischer Aufbau von Doctrine

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 Abfrage an eine MySQL und PostgreSQL Datenbank gesendet. Die Variable `$connection` enthält je eine initialisierte Verbindung zur entsprechenden Datenbank.

4 http://hibernate.org/
5 http://docs.doctrine-project.org/projects/doctrine1/en/latest/en/manual/acknowledgements.html

```php
<?php

$sql =
    'INSERT INTO students ('first_name', 'last_name', 'enrolment_number')
    VALUES ('Stefano', 'Kowalke', '12345');

// MySQLi
$result = mysqli_query($connection, $sql);

// PostgreSQL
$result = pg_query($connection, $sql);
```

Listing 1: Einfügen 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 schlussendlich gespeichert.

```php
<?php

$student = new Student();
$student->setFirstName('Stefano');
$student->setLastName('Kowalke');
$student->setEnrolmentNumber('12345');
$entityManager->persist($student);
$entityManager->flush();
```

Listing 2: Einfügen eines Studenten in die Datenbank mit ORM

Der Code in Listing 2 gibt keinen Rückschluss auf die darunter liegende 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 Listing 1 vergleichbar. Die Konvertierung der Abfrage in die verschiedenen SQL-Dialekt erfolgt durch Doctrine DBAL.

2.2.2 *DBAL*

Doctrine konvertiert das Schema anhand von sehr unterschiedlichen Merkmalen in das SQL des jeweiligen Database Management System (DBMS), das 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.

Datenbankschemata werden in Doctrine von der Klasse `Schema` repräsentiert. Im Beispiel wird zunächst eine Instanz dieser Klasse erstellt und anschließend eine neue Tabelle und mehrere Tabellenspalten mit unterschiedlichen Datentypen angelegt. In Zeile

30 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 aktuell benutzte DBMS.

```php
<?php

$schema = new \Doctrine\DBAL\Schema\Schema();
$beUsers = $schema->createTable('be_users‘);
$beUsers->addColumn('uid', 'integer',
  array('unsigned' => TRUE, 'notnull' => TRUE, 'autoincrement' => TRUE)
);
$beUsers->addColumn('pid', 'integer',
  array('unsigned' => TRUE, 'default' => '0', 'notnull' => TRUE)
);
$beUsers->addColumn('username', 'string',
  array('length' => 50, 'default' => '', 'notnull' => TRUE)
);
$beUsers->addColumn('password', 'string',
  array('length' => 100, 'default' => '', 'notnull' => TRUE)
);
$beUsers->addColumn('admin', 'boolean',
  array('default' => '0', 'notnull' => TRUE)
);
$beUsers->addColumn('history_data', 'text',
  array('length' => 16777215, 'notnull' => FALSE)
);
$beUsers->addColumn('ses_data', 'text',
  array('notnull' => FALSE)
);
$beUsers->setPrimaryKey(array('uid'));
$beUsers->addIndex(array('pid'), 'be_users_pid_idx');
$beUsers->addIndex(array('username'), 'be_users_username');

$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 einmal für PostgreSQL.

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

Listing 4: Das erstellte Schema als MySQL Abfrage

```
// Der Inhalt von $queries für PostgreSQL
CREATE TABLE be_users (
        uid serial NOT NULL,
        pid integer NOT NULL DEFAULT 0,
        username character varying(50) NOT NULL DEFAULT ''::character varying,
        password character varying(100) NOT NULL DEFAULT ''::character varying,
        admin boolean NOT NULL DEFAULT false,
        history_data text,
        ses_data text,
        CONSTRAINT be_users_pkey PRIMARY KEY (uid)
) WITH (
        OIDS=FALSE
);
```

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 der folgenden 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 Gedanken kommen, in dieser Spalte den kompletten Inhalt des Buches *The Hitchhiker's Guide to the Galaxy* von Douglas Adams speichern zu wollen, würde der Inhalt wie folgt aussehen: "Far out in the uncharted backwate" [Ada95, S. 3]. Der Rest des Textes würde abgeschnitten.

Bei numerischen Datentypen beschreibt der Wert allerdings die Anzeigenbreite - also die Anzahl der angezeigen Ziffern des gespeicherten Wertes. Sollte der in der Spalte gespeicherte Wert kleiner sein als die Länge der Anzeigenbreite, würden die restlichen Stellen nach links mit Leerzeichen aufgefüllt. Wurde die Option `ZEROFILL` gesetzt, würden die restlichen Stellen mit Nullen anstelle von Leerzeichen aufgefüllt. Der Wert beeinflusst in keinster 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 2.2 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 2.3 zeigt die Klassen mit einer Auswahl an Methoden.

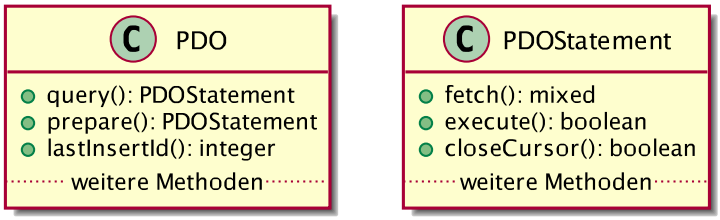


Abbildung 2.3: Die beiden PDO-Klassen

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 2.4). 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.

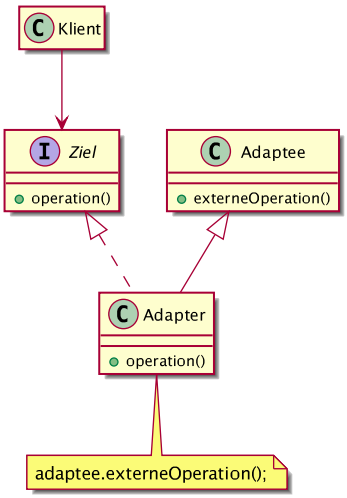


Abbildung 2.4: Das Adapter Entwurfsmuster in UML-Notation

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 2.5 zeigt die Implementation der Connection API, die diesem Muster folgt.

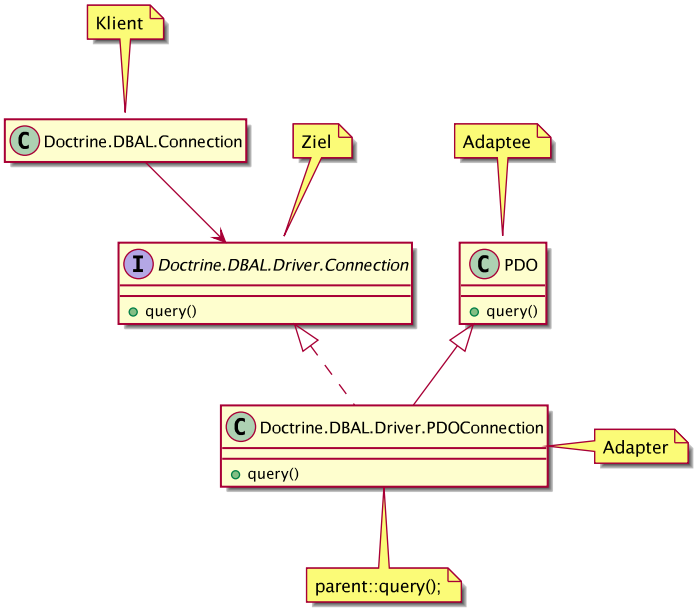


Abbildung 2.5: Aufbau des Connection-Objekts von Doctrine DBAL

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

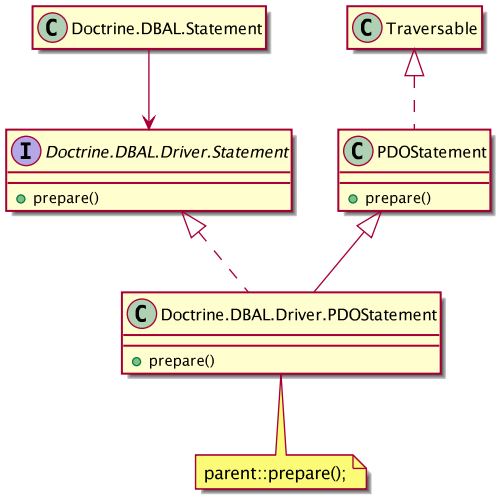


Abbildung 2.6: Aufbau des Statement-Objekts von Doctrine DBAL

Verbindung aufbauen

Über `Doctrine\DBAL\DriverManager::getConnection()` wird eine Verbindung angefordert. Dies wird in Listing 6 gezeigt.

```php
$config = new \Doctrine\DBAL\Configuration();

$connectionParams = array(
    'dbname' => 'hogwartsDB',
    'user' => 'snape',
    'password' => 'secret',
    'host' => 'localhost',
    'driver' => 'pdo_mysql',
);

$connection = \Doctrine\DBAL\DriverManager::getConnection(
                  $connectionParams,
                  $config
              );
```

Listing 6: Aufbau einer Datenbankverbindung mit Doctrine DBAL

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 `Logger`-Objekt 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   | Huffelpuff |
+----+------------+-----------+------------+
```

Tabelle 2.2: Auszug aus der Datenbanktabelle

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.

```php
$sql = 'SELECT last_name FROM students ORDER BY last_name';
// For MySQLi:
$result = mysqli_query($connection, $query);
while($row = mysqli_fetch_assoc($result)) {
    echo $row['last_name'] . ' ';
}

// PostgreSQL:
$result = pg_query($query);
while($row = pg_fetch_assoc($result)) {
    echo $row['last_name'] . ' ';
}
```

Listing 7: Ausgabe der Studierenden mit MySQL und PostgreSQL

In Doctrine DBAL enthält `Doctrine\DBAL\Connection` die Verbindung zur Datenbank. Um eine Abfrage 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 `foreach`-Schleife iteriert werden.

```php
$sql = 'SELECT last_name FROM students ORDER BY last_name';

$statement = $connection->query($sql);

foreach($statement as $row) {
  echo $row['last_name'] . ' ';
}
```

Listing 8: Einfache Datenbankabfrage mit Doctrine DBAL

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 Listing 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:

```
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  }
```

Listing 9: Steuerung der Formatierung der Ergebnismenge

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

- `PDO::FETCH_ASSOC` = `*_fetch_assoc()`[6]
- `PDO::FETCH_NUM` = `*_fetch_array()`
- `PDO::FETCH_ROW` = `*_fetch_row()`

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

Über das `Doctrine\DBAL\Statement`[7] werden weitere Möglichkeiten wie die Methoden `Doctrine\DBAL\Statement::fetch()` und `Doctrine\DBAL\Statement::fetchAll()` angeboten, um das Ergebnis zu erhalten.

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

Listing 10: Übergabe der Konstante an die fetch()-Methode

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 Statement wird zunächst an die Datenbank gesendet, wo der SQL-Parser die Struktur des Templates einmalig analysieren und vorkompiliert im Cache speichern kann. In einer zweiten Abfrage 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 Abfrage 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 Hut[8] hat bereits über die Häuser der Neuzugänge entschieden.

Die Daten der Studierenden liegen in einem assoziativen Array vor und können somit über eine `foreach`-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.

In Listing 11 wird bei jedem Durchlauf eine neue Abfrage mit den aktuellen Daten erzeugt und an die Datenbank geschickt. Für diese wiederkehrende Aufgabe bietet sich die Benutzung von Prepared Statements an, da pro Iteration lediglich die Werte angepasst werden müssen.

7 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 Abfrage 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 eine Verantwortlichkeit (engl. Single Responsibility Principle) haben darf. (vgl. [Mar08, S. 181])

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

```php
$students = array (
    array (
        'last_name' => 'Ellesmere',
        'first_name' => 'Corin',
        'house' => 1
    ),
    array (
        'last_name' => 'Tugwood',
        'first_name' => 'Havelock',
        'house' => 4
    ),
    array (
        'last_name' => 'Fenetre',
        'first_name' => 'Valentine',
        'house' => 3
    )
)

foreach ($students as $student) {
    $sql = 'INSERT INTO students (last_name, first_name, house)
      VALUES (' . $connection->quote($student['last_name']) .
        ',' . $connection->quote($student['first_name']) .
        ',' . $connection->quote($student['house']) . ')';

    $connection->query($sql);
}
```

Listing 11: INSERT Abfrage ohne Prepared Statements

Das folgende Listing 12 verwendet anstelle der eigentlichen Daten Fragezeichen als Platzhalter, 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.

```php
$statement = $connection->prepare(
  'INSERT INTO students (last_name, first_name, house)
    VALUES (?, ?, ?)');

foreach ($students as $student) {
    $statement->execute(
      array(
        $student['last_name'],
        $student['first_name'],
        $student['house']);
    );
}
```

Listing 12: Prepared Statements mit Positional Parameter

PDO bietet - im Gegensatz zu MySQLi - mit den *Named Parametern* 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 Listing 12 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 }
```

Listing 13: Prepared Statements mit Named Parameter

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, dass 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.

```php
$statement = $connection->prepare(
  'INSERT INTO students (last_name, first_name, house)
    VALUES (:lastname, :firstname, :house)');

foreach ($students as $student) {
    $statement->bindValue(':lastname', $student['first_name']);
    $statement->bindValue(':firstname', $student['last_name']);
    $statement->bindValue(':house', $student['house'], PDO::PARAM_INT);

    $statement->execute();
}
```

Listing 14

Die Methode `Doctrine\DBAL\Statement::bindParam()` unterscheidet sich dazu 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.

```php
$statement = $connection->prepare(
  'INSERT INTO students (last_name, first_name, house)
    VALUES (:lastname, :firstname, :house)');

$statement->bindParam(':lastname', $student['first_name']);
$statement->bindParam(':firstname', $student['last_name']);
$statement->bindParam(':house', $student['house'], PDO::PARAM_INT);

foreach ($students as $student) {
    $statement->execute();
}
```

SQL-Injections

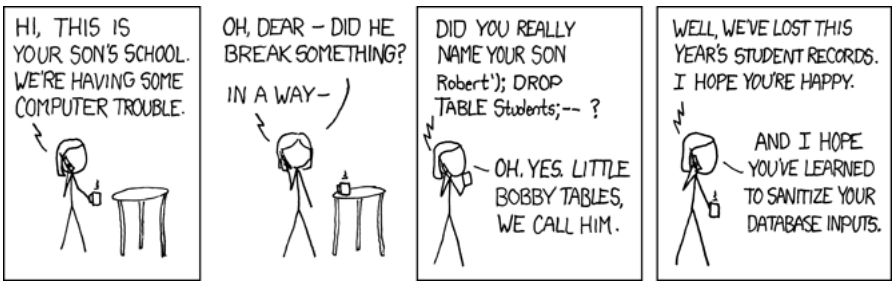


Abbildung 2.7: xkcd: Exploits of a mom[9]

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 Administrator der Webanwendung zurückliefert
- Daten in der Datenbank manipuliert
- oder die Datenbank ganz- oder teilweise löscht

Für ein kurzes Beispiel einer SQL-Injection soll ein Formular dienen, in dem nach den Nachnamen der Studierenden aus Hogwarts gesucht werden kann. Der gesuchte Datensatz wird ausgegeben wenn er gefunden wird, ansonsten erscheint eine entsprechende Meldung, dass nichts gefunden wurde. Der in das Eingabefeld 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-Abfrage dar.

```
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 15

Dieser Code beinhaltet zwei Fehler:

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

9 http://xkcd.com/327/

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. Dieses 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 Abfrage 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 kriminelle Absichten könnte mit der Suche nach einem Studenten mit dem Namen *O'Hara* die SQL-Injection auslösen. Die in diesem Fall an die Datenbank gesendete SQL-Abfrage `'SELECT * FROM students WHERE last_name = 'O'Hara';` würde wohl einen Fehler auslösen, da der Parser die Abfrage 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.

Die an `Doctrine\DBAL\Connection::query()` übergebenen SQL-Abfragen müssen durch `Doctrine\DBAL\Connection::quote()` maskiert werden. Traditionell wird dafür die PHP-Funktion `addslashes()` oder die jeweiligen Maskierungsmethoden der DBMS verwendet. Werden Prepared Statements genutzt, müssen die Benutzereingaben trotzdem überprüft, jedoch nicht mehr maskiert werden. Dies übernimmt der SQL-Parser, der lediglich 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-Abfragen in den SQL-Dialekt des Herstellers übersetzt.

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

Das folgende Beispiel zeigt die von MySQL unterstützte 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 16

Stattdessen muss die Abfrage 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 17

2.3 *Arbeitsweise*

Aufgrund seiner Aufgabe stellt der Prototyp einen tiefen Eingriff in die Architektur von TYPO3 dar. Vor diesem Hintergrund ist es unumgänglich, 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 allgemein, wenn die Codebasis in einer konsistenten Form formatiert wurde. Dies erhöht die Wartbarkeit und verbessert die Code Qualität.

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

In den TYPO3 Coding Guidelines (CGL)[10] 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 Klassendatei mit notwendigem Inhalt, der unabhängig von dem Zweck der Klasse vorhanden sein muss.

Der größte Teil der CGL behandelt die Formatierung verschiedener Sprachkonstrukte wie Schleifen, Arrays und Bedingungen.

Da das Regelwerk mit 28 Seiten recht umfangreich ausfällt, wird zur Überprüfung des Quellcodes das Programm PHP_CodeSniffer[11] in Zusammenhang mit dem entsprechenden Regelset für das TYPO3 Projekt [12] 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

10 http://docs.typo3.org/typo3cms/CodingGuidelinesReference/6.2/_pdf/manual.t3cgl-6.2.pdf
11 https://github.com/squizlabs/PHP_CodeSniffer
12 https://github.com/typo3-ci/TYPO3CMS

funktionieren. Um dies zu überprüfen muß die Funktionalität von TYPO3 in Verbindung mit der alten und der neuen 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 anderer 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 PHPUnit[13]. Es stellt das PHP Pendant von dem aus der Javawelt bekannten JUnit dar und wird von TYPO3 unterstützt. TYPO3 bringt selbst schon über 6000 UnitTests mit[14].

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

2.3.3 *Versionsverwaltung*

Als Versionsverwaltung wurde GIT[15] in Verbindung mit dem Code Hostingdienst GitHub[16] genutzt. Github dient zum einen als Backup im Falle eines Festplattenausfalls und zum anderem als späterer Anlaufpunkt der Extension für Interessierte.

2.3.4 *Composer*

Composer[17] ist ein *Dependency Manager* für PHP. Er installiert externe Bibliotheken, die über eine Konfigurationsdatei als Abhängigkeit festgelegt wurden.

2.3.5 *IDE*

Als Editor wurde die IDE PHPStorm verwendet, die über Autovervollständigung von Variablen, Methoden und Klassenverfügt und den Programmierer bei der Erstellung von Klassen durch Templates unterstützt.

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 dessen 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 neu geladen, bleibt das Programm an dem gesetzten Breakpoint stehen.

13 http://phpunit.de/
14 https://travis-ci.org/TYPO3/TYPO3.CMS/builds/23070563
15 http://git-scm.com/
16 http://github.com
17 getcomposer.org

3

AKTUELLE SITUATION

3.1 *Die native Datenbank API*

n Kapitel 2.1.1 wurde bereits darauf hingewiesen, dass die einheitliche Datenbank API von der Klasse `\TYPO3\CMS\Core\Database\DatabaseConnection` bereitgestellt wird. Über die globale Variable `$GLOBALS['TYPO3_DB']` kann darauf zugegriffen werden.

```
1  $GLOBALS['TYPO3_DB']->exec_UPDATEquery(
2    $this->user_table,
3    $this->userid_column . '=' .
4      $this->db->fullQuoteStr(
5        $tempuser[$this->userid_column],
6        $this->user_table
7      ),
8    array($this->lastLogin_column => $GLOBALS['EXEC_TIME'])
9  );
```

Listing 18: Aktualisierung des Zeitpunkts des letzten Logins

Durch die Nutzung der Datenbank API wird zum einen die Integrität der Daten sichergestellt und es werden die PHP-Datenbankfunktionen abstrahiert. Somit kann die Implementation der API-Methoden angepasst werden, ohne die API selbst zu verändern. Auf diese Weise wurde die Umstellung von MySQL auf die MySQLi durchgeführt.[1]

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

1. Methoden die SQL-Abfragen generieren
2. Methoden die SQL-Abfragen gegen die Datenbank ausführen
3. Methoden die auf der Ergebnismenge agieren
4. Administrative Methoden
5. Hilfsmethoden

1 http://bit.ly/typo3cms-switch-from-mysql-to-mysqli

3.1.1 *Generierende Methoden*

Die erste Gruppe besteht aus Methoden, die anhand der übergebenen Parameter einen SQL-Abfrage generieren. Damit werden die typischen CRUD-Operationen 2 abgebildet.

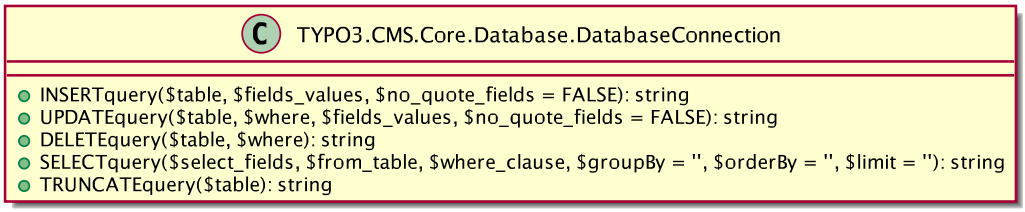


Abbildung 3.1: Methoden zum Erzeugen von SQL-Abfragen

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

```
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(
5        'pages',
6        'sys_file_reference'
7      )
8  );
```

Listing 19: Löschen eines Datensatzes aus einer Tabelle

3.1.2 *Ausführende Methoden*

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

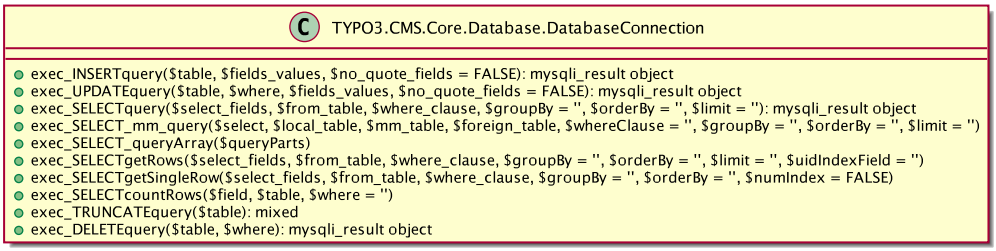


Abbildung 3.2: Methoden zum Ausführen von SQL-Abfragen

3.1.3 *Methoden zur Manipulation der Ergebnismenge*

In der dritten 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,
- die für die Fehlerbehandlung genutzt werden können,
- sowie Methoden, die Informationen über die Ergebnismenge bereitstellen

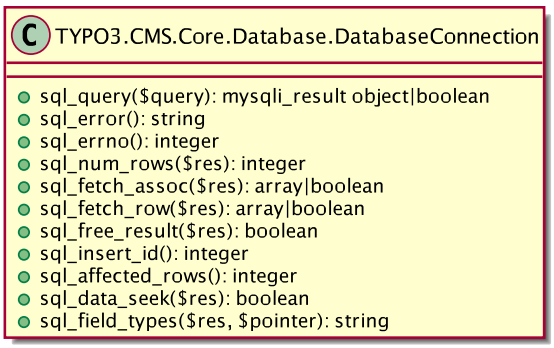


Abbildung 3.3: Methoden zur Verarbeitung der Ergebnismenge

3.1.4 *Administrative Methoden*

Die nächste 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

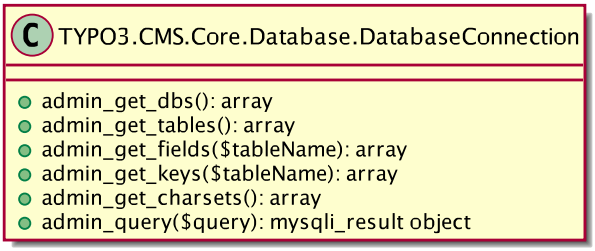


Abbildung 3.4: Administrativen Methoden

3.1.5 *Hilfsmethoden*

Die letzte Gruppe besteht aus Hilfsmethoden, die genutzt werden um

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

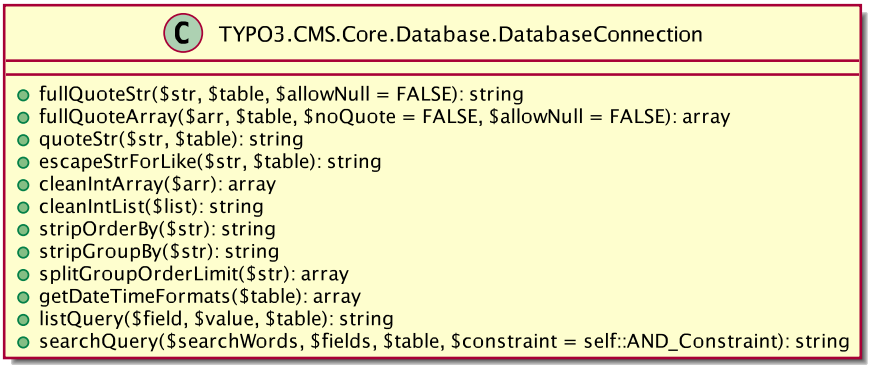


Abbildung 3.5: Hilfsmethoden

3.2 *Prepared Statements*

Seit TYPO3 CMS 4.5 können Prepared Statements für `SELECT` Abfragen verwendet werden. TYPO3 CMS unterstützt sowohl *Posistional 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 20: 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.

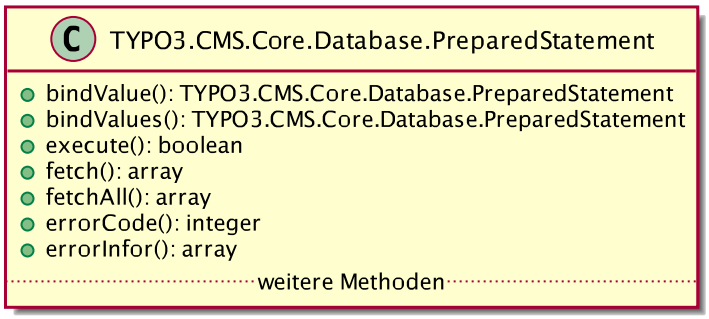


Abbildung 3.6: Die Klasse PreparedStatement mit ausgewählten Methoden

3.3 *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 Nutzergruppen
- 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 einer 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, die 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.

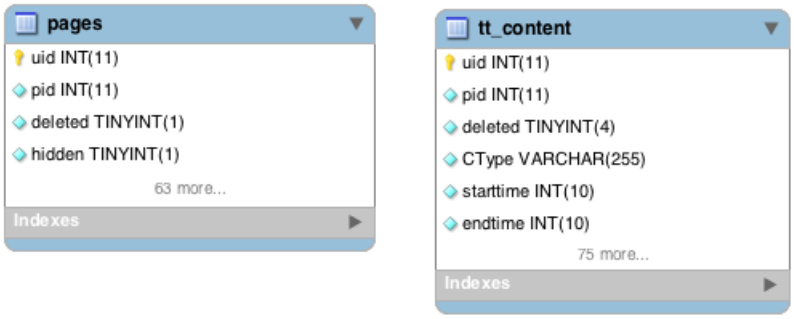


Abbildung 3.7: Die Tabellen pages und tt_content

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 21 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 Abfrage 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 21: Abrufen von Unterseiten einer Seite

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

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

Listing 22: 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-Abfragen 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 zu Gruppe von den Tabellen `fe_users` und `fe_groups` beziehungsweise `be_users` und `be_groups` abgebildet.

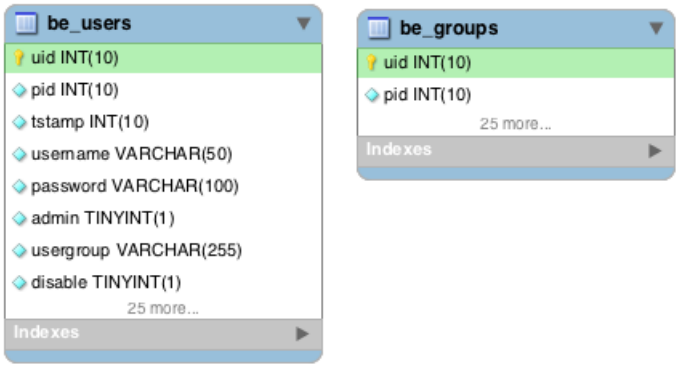


Abbildung 3.8: Die Tabellen be_users und be_groups

In Abbildung 3.8 fällt der Datentyp der Spalte `usergroup` auf, der die ID der Gruppe des Benutzers speichert. Er wurde als Typ `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	deleted
1	0	1191353353	admin	secret	1		0
2	0	1281556682	snape	secret	1	43	0
3	0	1191353353	hagrid	secret	0	5,32,43	0

Tabelle 3.1: Auszug aus der be_users Tabelle

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-Abfrage erhöht.

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

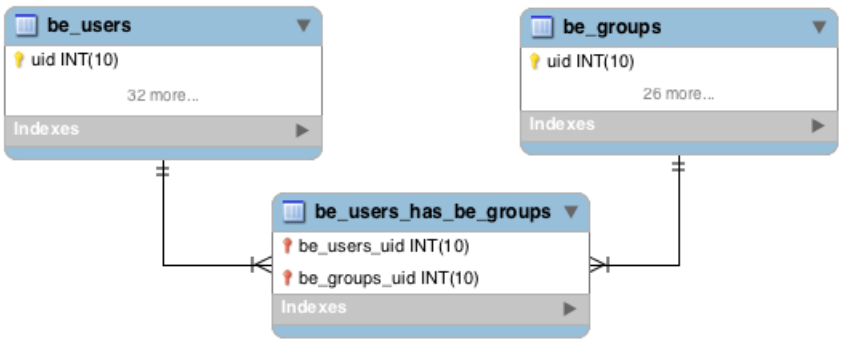


Abbildung 3.9: Normalisierung über Many-to-Many Tabelle

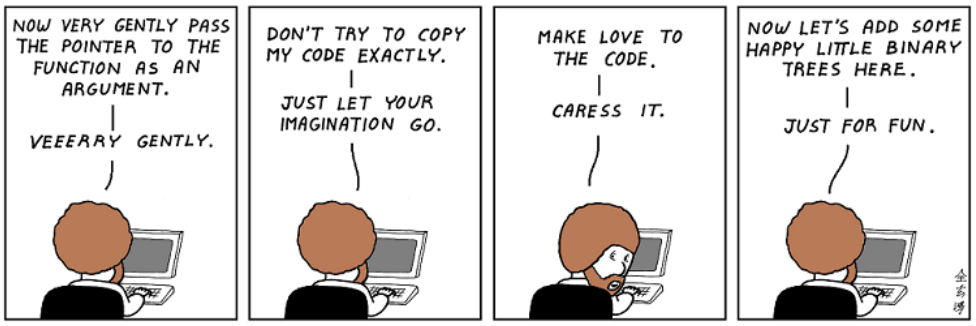
be_users_uid	be_groups_uid
2	34
3	5
3	32
3	43

Tabelle 3.2: Die MM-Tabelle für be_users

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

4

PROTOTYPISCHER NACHWEIS DER HERSTELLBARKEIT



4.1 *Konzeption des Prototypen*

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

- 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 gegebenenfalls 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). Dazu muss eine Übersicht angefertigt werden, die die neuen Namen den alten Namen gegenüberstellt. Diese befindet sich im Anhang.

- 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 Abfragen 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

4.2 *Vorbereitung*

4.2.1 *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. *GIT* wurde verwendet, damit der Code einfacher aktualisierbar war und eigene Änderungen daran nachvollzogen und dokumentiert werden konnten.

Abbildung 4.1 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 4.1: 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.

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.

Schritt 1 - Systemcheck

Im ersten Schritt (Abb.: 4.2a) 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.

Schritt 2 - Eingabe der Datenbankdaten

Im zweiten Schritt (Abb.: 4.2b) werden die Benutzerdaten für die Datenbank eingegeben. Es kann zwischen einer Port- oder Socket-basierten Verbindung ausgewählt werden.

Wenn anstelle von MySQL ein alternatives DBMS genutzt werden soll, kann über die Schaltfläche am Ende des Formulars die Systemextension DBAL installiert werden.

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.: 4.2c). Über die Auswahl kann eine leere Datenbank festgelegt werden. Alternativ kann über das Eingabefeld eine zu erstellende Datenbank angegeben werden. Mit dem Absenden des Formulars werden die Basistabellen in der Datenbank angelegt.

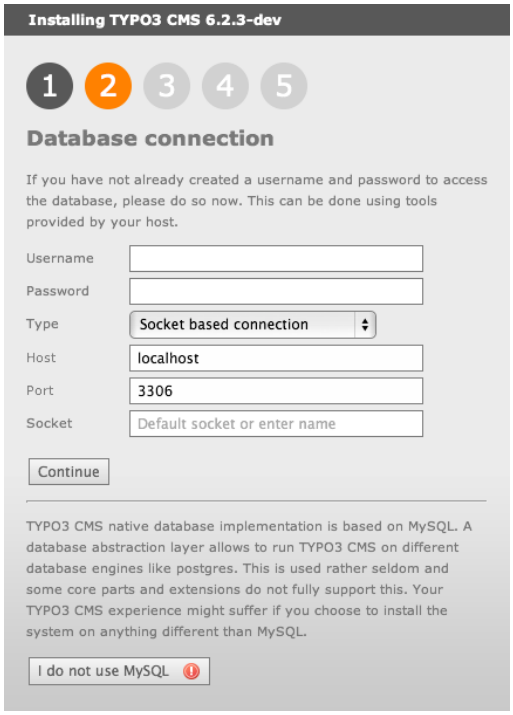
Schritt 4 - Einrichten eines TYPO3 Administrators

Im 4. Schritt (Abb.: 4.2d) der Installation wird ein Administrator eingerichtet und es kann ein Name für die Seite vergeben werden.

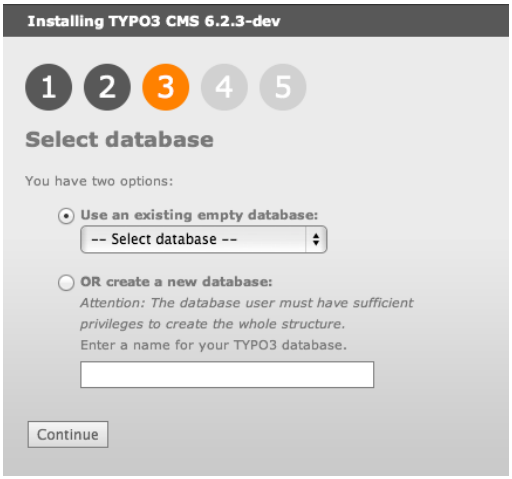
Schritt 5 - Abschluss der Installation

Danach ist die Installation abgeschlossen und über die Schaltfläche kann das Backend aufgerufen werden (Abb.: 4.2e)

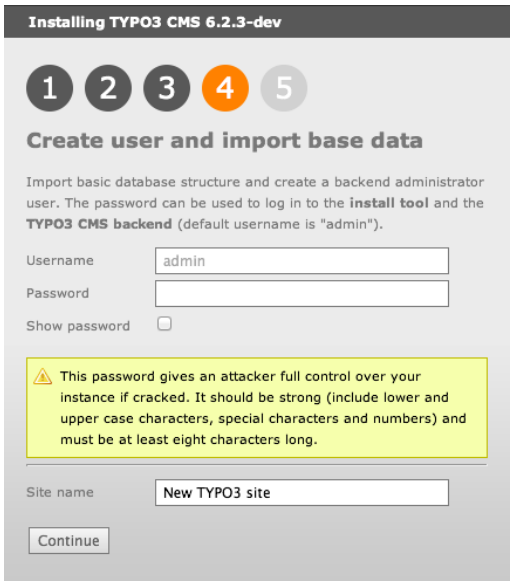
(a) Installation TYPO3 CMS - 1. Schritt



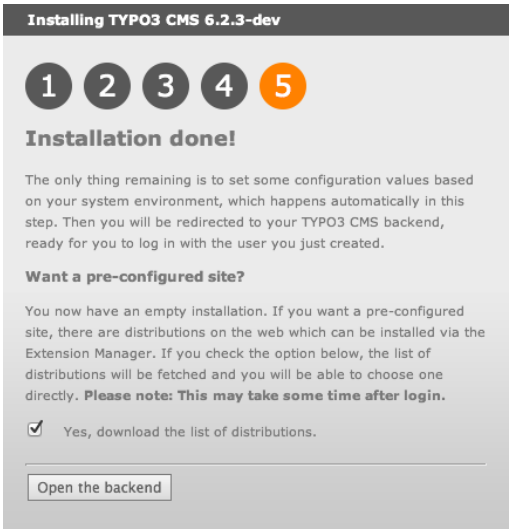
(b) Installation TYPO3 CMS - 2. Schritt



(c) Installation TYPO3 CMS - 3. Schritt



(d) Installation TYPO3 CMS - 4. Schritt



(e) Installation TYPO3 CMS - 5. Schritt

Abbildung 4.2: Installation von TYPO3 CMS

4.2.2 *Implementation von Unit Tests der alten Datenbank API*

Um zu gewährleisten, dass TYPO3 CMS sowohl mit der alten API - die von dem Prototypen zur Verfügung gestellt wird - als auch mit der neuen API kompatibel ist, müssen Unit Tests für die alte Datenbank API geschrieben werden.

Zur Ausführung der Unit Tests wird die Extension *PHPUnit* benötigt, welche das gleichnamige Testing Framework *PHPUnit*[1] zur Verfügung stellt und einen graphischen Testrunner im Backend mitbringt. Sie wird über den Extension Manager installiert.

Die alte Datenbank API verfügt zu dem Zeitpunkt der Erstellung des Prototypen über 40 Tests mit 49 Assertions, welche jedoch lediglich Hilfsmethoden testen. Im Laufe der Arbeit wurden 68 Tests implementiert, die alle Methoden testen. Die Abbildung 4.3 zeigt die Ausführung der von TYPO3 CMS mitgelieferten Unit Tests; Abbildung 4.4 zeigt die Ausführung der neu implementierten sowie der alten Tests.

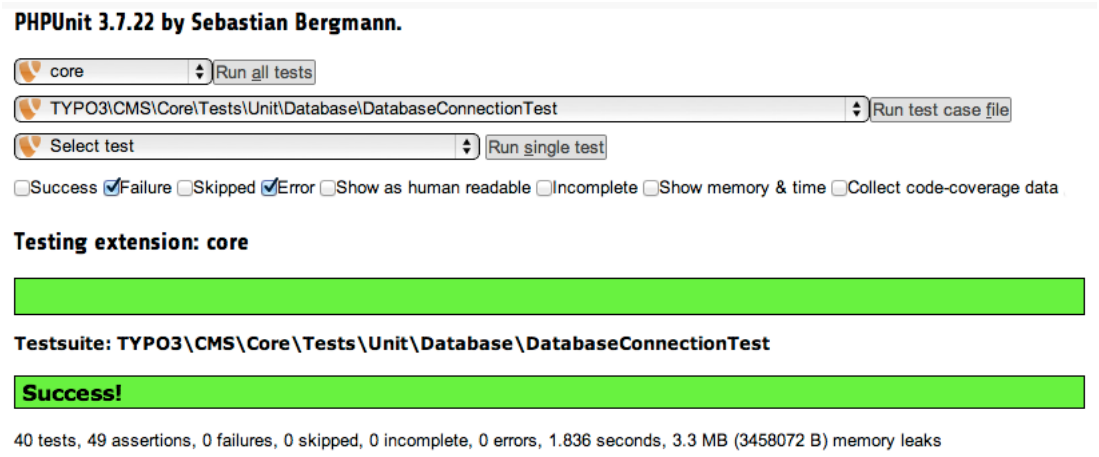


Abbildung 4.3: Ausführung der vorhandenen Unit Tests für die alte Datenbank API

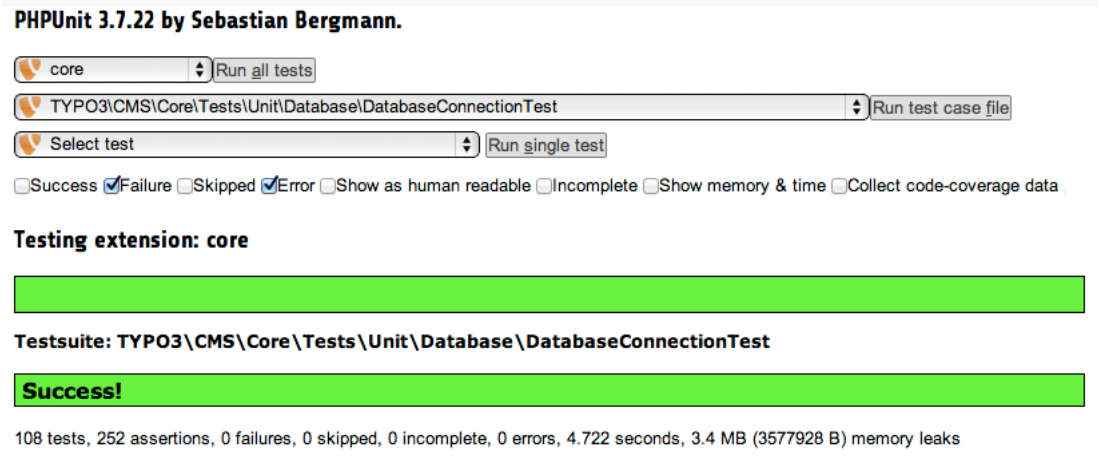


Abbildung 4.4: Ausführung der vorhandenen und hinzugefügten Unit Tests für die alte Datenbank API

1 http://www.phpunit.de

4.3 *Erstellung des Prototypen*

4.3.1 *Die Grundstruktur*

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 23: 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"
  },
  "mininum-stability": "dev",
}
```

Listing 24: 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 Abfragen 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:. 4.5)

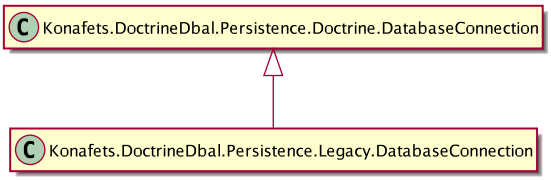


Abbildung 4.5: Alte API-Klasse erbt von neuer API-Klasse

- die Klasse der alten API per XCLASS in der Datei `ext_localconf.php` registriert (Listing 25).

```
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'
9        => 'Konafets\\DoctrineDbal\\Persistence\\Legacy\\DatabaseConnection'
10   );
```

Listing 25: 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.

4.3.2 *Refactoring der alten API*

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 hat sie

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

Der 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/Exeptions/` erstellt. Listing 26 zeigt die Methode nach dem Refactoring; Listing 27 zeigt den Aufruf der neuen Methode. Die originale Implementation der Methode ist in `typo3/sysext/core/Classes/Database/DatabaseConnection.php` ab Zeile 1578 zu finden.[2]

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

```php
public function connectDatabase($isInitialInstallationInProgress = FALSE) {
    // Early return if connected already
    if ($this->isConnected) {
        return;
    }

    if (!$isInitialInstallationInProgress) {
        $this->checkDatabasePreconditions();
    }

    try {
        $this->link = $this->getConnection();
    } catch (\Exception $e) {
        throw new ConnectionException($e->getMessage());
    }

    $this->isConnected = $this->checkConnectivity();

    if (!$isInitialInstallationInProgress) {
        if ($this->isConnected) {
            $this->initCommandsAfterConnect();
            $this->selectDatabase();
        }

        $this->prepareHooks();
    }
}
```

Listing 26: connectDatabase() nach dem Refactoring

```php
public function connectDB(...) {
    ...
    $this->connectDatabase();
}
```

Listing 27: connectDB() delegiert an die neue API-Methode

4.3.3 *Herstellen einer Verbindung*

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 der 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, indem ein eigenes `PDO`-Objekt mit dem Konstrukturparameter `\PDO::ATTR_PERSISTENT => true` erstellt und in dem Konfigurationsarray gespeichert wurde, welches im Anschluß an `\Doctrine\DBAL\DriverManager::getConnection()` übergeben wurde (siehe Listing 28).

```php
protected $connectionParams = array(
    'dbname'   => '',
    'user'     => '',
    'password' => '',
    'host'     => 'localhost',
    'driver'   => 'pdo_mysql',
    'port'     => 3306,
    'charset'  => 'utf8',
);
```

Listing 28: Das Konfigurationsarray für Doctrine DBAL

Zudem muß Doctrine DBAL der Datentyp `Enum` bekannt gemacht 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.

4.3.4 *Implementation der grundlegenden Methoden*

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 Abfragen an die Datenbank dar. Sie kapselt die gleichnamige Methode `\Doctrine\DBAL\Connection::query()`.

3 http://bit.ly/1sOXU1q

```php
protected function query($query) {
    if (!$this->isConnected) {
        $this->connectDatabase();
    }

    $stmt = $this->link->query($query);

    return $stmt;
}
```

Listing 29: getConnection() der neuen API

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.

```php
public function fetchAssoc($stmt) {
    if ($this->debugCheckRecordset($stmt)) {
        return $stmt->fetch(\PDO::FETCH_ASSOC);
    } else {
        return FALSE;
    }
}
```

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.

```php
public function listDatabases() {
    ...
    $databases = $this->schemaManager->listDatabases();
    ...
    return $databases;
}
```

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 weiteren 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. Sicherer ist jedoch von Anfang die Benutzung von Prepared Statements, welche von TYPO3 CMS seit Version 6.2 in Form eines Wrap-

pers um die Prepared Statements von MySQLi angeboten werden. Sie besitzen die gleichen API, wie sie in Kapitel 2.2.4 vorgestellt wurden.

Durch den Aufruf der Methode `$stmt = $GLOBALS['TYPO3_DB']->prepare($sql)` wird zunächst das Objekt vom Typ `\TYPO3\CMS\Core\Database\PreparedStatement` erstellt, dem der in `$sql` gespeicherte Prepared Statement übergeben wird. Ein Aufruf von `$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-Abfrage 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.

Zur Umstellung auf Doctrine wurde die Datei `PreparedStatement.php` TYPO3 CMS nach `doctrine_dbal/Classes/Persistence/Doctrine/` kopiert. Es wurden die eigens verwendeten Konstanten auf PDO-Konstanten umgemappt und die Transformation in *Positional Parameter* wurde entfernt, Doctrine DBAL 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 }
```

4.4 *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 4.2.1 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 Model-View-Controller (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/Privat/` 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.
- Partials stellen wiederkehrende Elemente dar. Sie können in Layout- und Templatedateien eingebunden werden. Die Schaltfläche *I do not use MySQL* aus Abbildung 4.2b in Kapitel 4.2.1 ist ein Beispiel für ein Partials.

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 30
- Es wurden die Partials `LoadDoctrineDbal.html` und `UnloadDoctrineDbal.html` zur De- und Installation des Prototypen sowie das Partial `DoctrineDbalDriverSelection.html` für die Auswahl des Datenbanktreibers erstellt und 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 anzeigt.
- 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 leerer 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 Mananger 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.

```php
// Bootloader.php
public function initializeClassLoader() {
    /** Composer loader */
    require_once PATH_typo3conf . 'ext/doctrine_dbal/vendor/autoload.php';

    $classLoader = new ClassLoader($this->applicationContext);
    ...
}
```

Listing 30: Einbinden des vom Composer erstellten Autoloaders

4.4.1 *Doctrines Schemarepräsentation*

Während der Installation werden die initialen Datenbanktabellen angelegt. Die dafür notwendigen SQL-Abfragen halten die Extensions in `*.sql`-Dateien vor. Dabei handelt es sich um einfache Textdateien, die aus ein oder mehreren `CREATE TABLE` Abfragen bestehen. Diese werden von `TYPO3\CMS\Core\Database\SqlParser` geparst, 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:felogin` erwähnt, die beide die Tabelle `fe_users` anlegen.

```sql
# ext:felogin
CREATE TABLE fe_users (
        felogin_redirectPid  tinytext,
        felogin_forgotHash  varchar(80) default ''
);

# ext:frontend
CREATE TABLE fe_users (
        uid int(11) unsigned NOT NULL auto_increment,
        pid int(11) unsigned DEFAULT '0' NOT NULL,
        tstamp int(11) unsigned DEFAULT '0' NOT NULL,
        username varchar(50) DEFAULT '' NOT NULL,
        password varchar(100) DEFAULT '' NOT NULL,
        usergroup tinytext,
        …
);
```

Listing 31

Der Importvorgang erfolgt durch die Methode `importDatabaseData()` aus der Klasse `\TYPO3\CMS\Install\Controller\Action\Step\DatabaseData`. Dabei ermittelt die Methode `\TYPO3\CMS\Install\Service\SqlExpectedSchemaService::getTablesDefinitionString()` anhand der `*.sql`-Dateien den Soll-Zustand. Die Methode `getFieldDefinitions_database()` aus der Klasse `\TYPO3\CMS\Install\Service\SqlSchemaMigrationService` ermittelt den Ist-Zustand. Beide Zustände werden anschließend durch die Methoden `getDatabaseExtra()` und `getUpdateSuggestions()` derselben Klasse verglichen. Die genauere Analyse der Methoden würde für die Arbeit zu weit führen, deswegen wird hier darauf verzichtet. Anschließend wird die Differenz in Form von SQL-Abfragen 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 `\Doctrine\DBAL\Schema\Schema`-Objekte, da diese die notwendige Abstraktion bieten und von Doctrine DBAL anhand der verwendeten Plattform in die entsprechenden SQL-Abfragen konvertiert werden können.

Die `*.sql`-Dateien folgender Systemextensions wurden umgewandelt:

- 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-Zustands wurde der Klasse `SqlExpectedSchemaService` die Methode `getTablesDefinitionAsDoctrineSchemaObjects()` hinzugefügt, die die vorhandenen `Schema.php`-Dateien sucht und per `require` dem PHP-Skript zur Verfügung stellt. 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 dynamisch nach dem

Muster `cf_cache_<tabellenname>` beziehungsweise `cf_cache_<tabellenname>_tags` erstellt. Die dazu implementierten Klasse `\TYPO3\CMS\Core\Cache\Schema\Typo3DatabaseBackendCacheSchema` und `\TYPO3\CMS\Core\Cache\Schema\Typo3DatabaseBackendTagsSchema` dienen dabei als Templates, denen bei der Instanziierung der *<tabellenname>* mitgegeben wird. Sie erstellen intern ein Objekt vom Typ `\Doctrine\DBAL\Schema\Schema`, welches schließlich die ensprechende 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 `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 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` per XCLASS registriert. Sie wird um die Methode `getCurrentSchemaFromDatabase()` erweitert, die den *Schema Manager* von Doctrine nutzt, um den Zustand der Datenbank unabhängig vom DBMS abzufragen. Der 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.

4.4.2 *Installation des Prototypen*

Schritt 1 - Systemcheck

Der erste Schritt entspricht dem aus Kapitel 4.2.1

Schritt 2 - Eingabe der Datenbankdaten

Im zweiten Schritt (Abb.: 4.6a) 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 4.6b zeigt die Felder für MySQL.

Schritt 3 - Auswahl der Datenbank

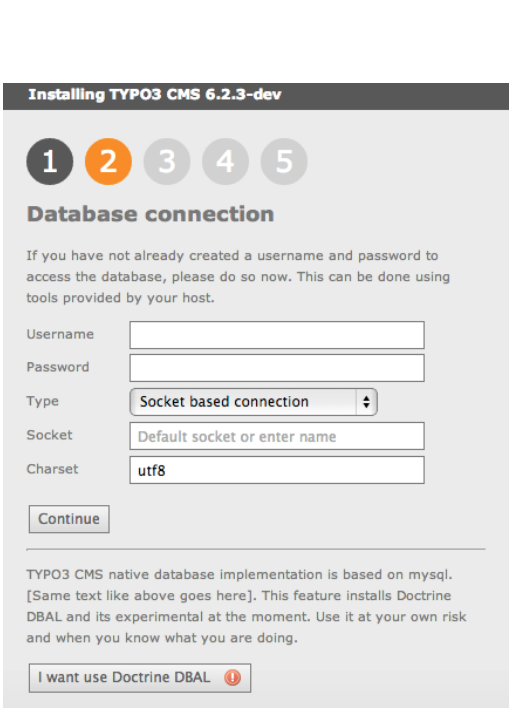
Nachdem die Verbindungsdaten eingegeben wurden, versucht TYPO3 CMS eine Verbindung zum DBMS zu etablieren. Gelingt dies, werden alle verfügbaren Datenbanken abgefragt und aufgelistet (Abb.: 4.2c). Über die Auswahl kann eine leere Datenbank festgelegt werden. Alternativ kann über das Inputfeld eine zu erstellende Datenbank angegeben werden. Mit dem Absenden des Formulars werden die Basistabellen in der Datenbank angelegt.

Schritt 4 - Einrichten eines TYPO3 Administrators

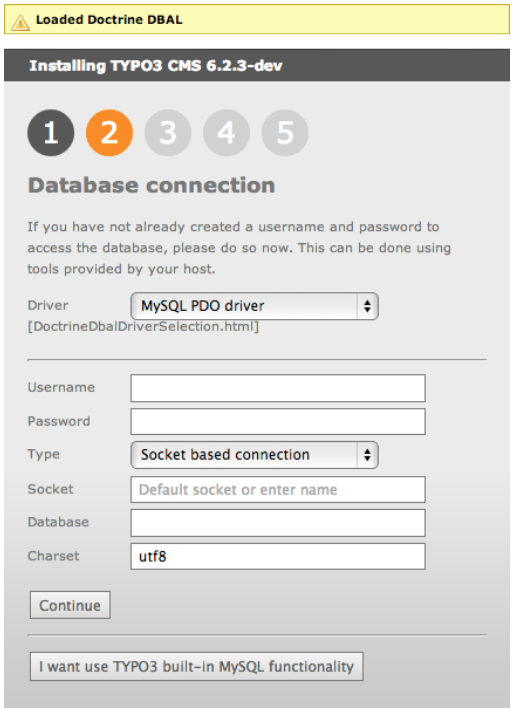
Der Schritt entspricht dem aus Kapitel 4.2.1

Schritt 5 - Abschluß der Installation

Der Schritt entspricht dem aus Kapitel 4.2.1



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



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

Abbildung 4.6: 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.

4.5 *Umstellen der Query-Methoden auf Doctrine DBAL und Prepared Statements*

In Kapitel 3 zur Datenbank API wurde bereits die Unterteilung in Generierende und Ausführende SQL-Methoden erläutert. Die Generierung der Abfrage erfolgt anhand der Parameter, die – zusammen mit den entsprechenden SQL-Abfragen – in eine Zeichenkette umgewandelt und zurückgegeben werden.

```
1  public function UPDATEquery(
2    $table,
3    $where,
4    array $fields_values,
5    $no_quote_fields = FALSE
6    ) {
7    ...
8      $fields = array();
9      if (is_array($fields_values) && count($fields_values)) {
10       // Quote and escape values
11       $nArr = $this->fullQuoteArray(
12                 $fields_values,
13                 $table,
14                 $no_quote_fields,
15                 TRUE
16               );
17       foreach ($nArr as $k => $v) {
18         $fields[] = $k . '=' . $v;
19       }
20     }
21
22     $query = 'UPDATE ' . $table . ' SET ' . implode(',', $fields) .
23         ((string)$where !== '' ? ' WHERE ' . $where : '');
24     ...
25     return $query;
26   }
27 }
```

Listing 32: Original UPDATEquery der alten Datenbank API

Doctrine DBAL bietet mit dem `QueryBuilder` eine Klasse an, mit der die Formulierung von SQL-Abfragen abstrahiert werden kann. Die Generierung der SQL-Abfrage aus Listing 32 konnte durch die Nutzung des `QueryBuilders` 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 33: Generierung der SQL-Abfrage 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 34), wird für die neue Datenbank API eine Abstraktion zur Formulierung von SQL-Abfragen nach dem Vorbild des `QueryBuilders` angestrebt (siehe Listing 35). Wie zu sehen ist, werden dort auch die logischen Ausdrücke abstrahiert.

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

Listing 34

```
protected function resetStageOfElements($stageId) {
  ...
      $dbh = $GLOBALS['TYPO3_DB'];
      $expr = $dbh->expr();
      $query = $dbh->createUpdateQuery();

      $query->update($tcaTable)
        ->set(
          't3ver_stage',
          \TYPO3\CMS\Workspaces\Service\StagesService::STAGE_EDIT_ID)
        ->where(
          $expr->equals(t3ver_stage, $stageId),
          $expr->greaterThan(t3ver_wsid, 0),
          $expr->equals(pid, -1)
        )
      );

      $query->execute();
 ...
}
```

Listing 35: 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 Abfragen 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 *Fassade* dar und bietet nach außen lediglich eine Teilmenge der von `QueryBuilder` zur Verfügung gestellten Methoden.

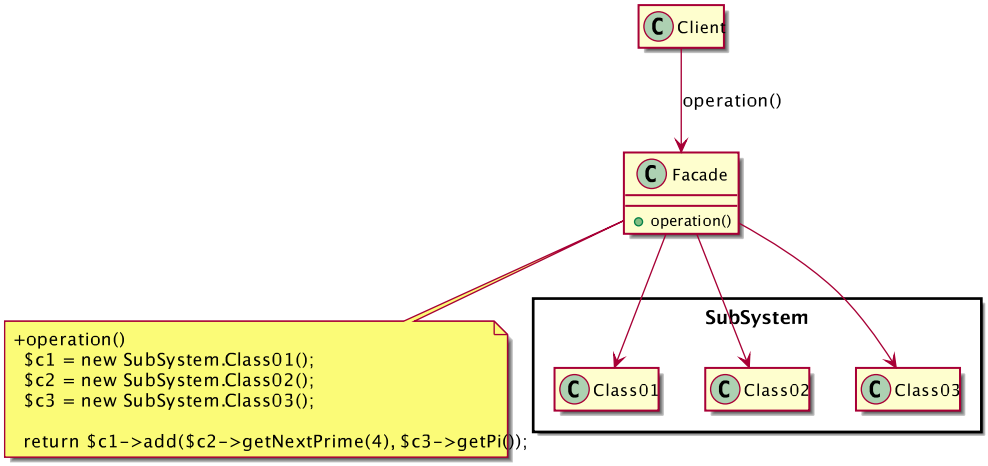


Abbildung 4.7: Schematischer Aufbau des Fassade-Entwurfsmusters

Das folgende UML-Diagramm zeigt anhand der `TruncateQuery`- und `UpdateQuery`-Objekte die fertige Hierachie. 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-Abfrage konvertiert und ausgeführt werden kann.

Das Interface wird um ein Interface erweitert, welches 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.

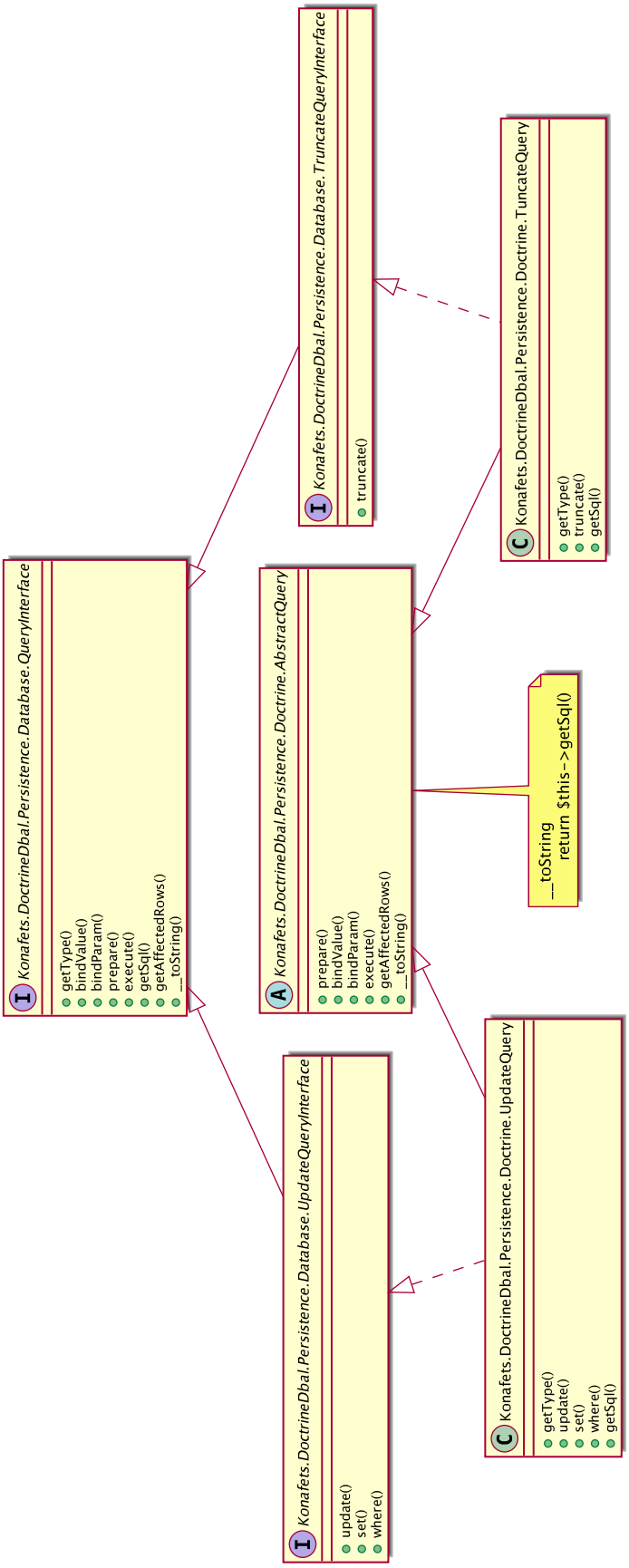


Abbildung 4.8: Aufbau der Query-API

Der Aufbau ist an die Implementation der Datenbank API von EzPublish angelehnt, die von Benjamin Eberlei eingeführt wurde.[5] Während die Erzeugung der SQL-Abfragen 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 36: 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.

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

```
public function createUpdateQuery() {
  if (!$this->isConnected) {
    $this->connectDatabase();
  }

  return GeneralUtility::makeInstance(
            '\\Konafets\\DoctrineDbal\\Persistence\\Doctrine\\UpdateQuery',
            $this->link
          );
}
```

Listing 37: Die Erzeugung eines UpdateQuery-Objekts

Nun konnte die Abhängigkeit zu Doctrine DBAL durch den direkten Aufruf des `QueryBuilders` aus Listing 33 entfernt werden:

```
public function UpdateQuery(...) {
...
  $query = $this->createUpdateQuery()
    ->update('pages')
    ->set('title', 'Foo')
    ->where('id = 1')
    ->getSQL();
...
}
```

Listing 38

Zur Abstraktion der logischen Ausdrücke aus dem Codebeispiel 35 wurde die Klasse `Expression` implementiert, die ebenfalls eine Fassade darstellt - dieses Mal vor der Klasse `\doctrine\DBAL\Query\ExpressionBuilder`. Das Namensschema folgt der Extbase-API.

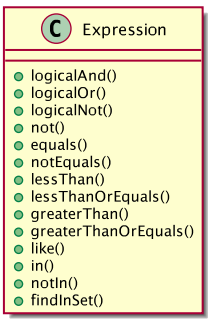


Abbildung 4.9: Die Klasse Expression

Die Query-API kann nun, wie es in Listing 35 gezeigt, verwendet werden.

Aus der Datei `typo3/sysext/core/Classes/Authentication/AbstractUserAuthentication.php` wurde die Methode `fetchUserSession()` ausgewählt. An ihr soll die Migration der alten API auf die neue API unter Verwendung des `UpdateQuery` gezeigt werden. Die Methode aktualisiert die Session für den angemeldeten Backendbenutzer und wird bei jedem Aufruf des Backends aufgerufen. Dadurch ist ist sehr gut überprüfbar, ob die Migration wie erwartet funktioniert. Die Methode erzeugt folgende SQL-Abfrage:

```
UPDATE be_sessions
SET ses_tstamp = '1400788819'
WHERE ses_id='c2c75…'
AND ses_name='be_typo_user'
```

Listing 39: Die von fetchUserSession erzeugte SQL-Abfrage

Die Werte in der `WHERE`-Bedingung werden durch `fullQuoteStr()` maskiert, um SQL-Injections zu verhindern.

```
public function fetchUserSession($skipSessionUpdate = FALSE) {
...
  if ($timeout > 0 && $GLOBALS['EXEC_TIME'] < $user['ses_tstamp'] + $timeout) {
    if (!$skipSessionUpdate) {

      $this->db->exec_UPDATEquery(
        $this->session_table,
        'ses_id=' . $this->db->fullQuoteStr(
          $this->id,
          $this->session_table
        ) .
          ' AND ses_name=' . $this->db->fullQuoteStr(
            $this->name,
            $this->session_table
          ),
        array('ses_tstamp' => $GLOBALS['EXEC_TIME']));

      // Make sure that the timestamp is also updated in the array
      $user['ses_tstamp'] = $GLOBALS['EXEC_TIME
    }
  } else {
    // Delete any user set...
    $this->logoff();
  }'];
…
}
```

Listing 40: 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 `FluentInterfaces` 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 `Fluent Interface` durch die Zurückgabe des eigenen Objekts in der Methode mittels `return $this`.

```
...
 if ($timeout > 0 && $GLOBALS['EXEC_TIME'] < $user['ses_tstamp'] + $timeout) {
   if (!$skipSessionUpdate) {

     $query = $this->db->createUpdateQuery();
     $query->update($this->session_table)
       ->set('ses_tstamp', $GLOBALS['EXEC_TIME'])
       ->where(
         $query->expr->equals('ses_id', $this->db->quote($this->id)),
         $query->expr->equals('ses_name', $this->db->quote($this->name))
       )->execute();
   }
 } else {
   // Delete any user set...
   $this->logoff();
 };}
...
)
```

Listing 41: fetchUserSession() nach der Migration mit manueller Maskierung

Auch hier wurde die Maskierung manuell durch `DatabaseConnection::quote()` vorgenommen. In Kapitel 2.2.4 wurde auf die Gefahren von SQL-Injections hingewiesen und wie diese durch Prepared Statements (siehe Kaptiel 2.2.4) verhindert werden können. Das nächste Listing zeigt wie diese mit der API genutzt werden können.

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

Listing 42: 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, das sich in `$query` befindet. Analog dazu existiert die Methode `bindParam()`, die nach dem Vorbild der gleichnamigen Methode von Doctrine DBAL funktioniert. Die erzeugte SQL-Abfrage 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-Abfrage 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 empfielt 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. In der neuen Datenbank API wurden Methoden gleichen Namens implementiert, die die Abfrage direkt an die Doctrine Methoden weiterreichen. Die

neuimplementierten Methoden `executeDeleteQuery()`, `executeInsertQuery()` und `executeUpdateQuery()` verfolgen ebenfalls diesen Ansatz, wurden jedoch als Ersatz zu den `exec_*`-Methoden der alten API konzipiert. Diese konnten jedoch ohne tiefergreifende 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.

FAZIT

In der Thesis wurden die Fragen nach der Intergrierbarkeit von Doctrine DBAL in TYPO3 CMS bei gleichzeitiger Beibehaltung der Kompatibilität zu der existierenden Datenbank API aufgeworfen. Der zu diesem Zweck erstellte Prototyp hat die Machbarkeit bewiesen. Es konnte gezeigt werden, dass Doctrine DBAL auf unterschiedlichen Ebenen integrierbar ist. Angefangen bei der Erstellung der Verbindung zur Datenbank über die Nutzung des Query Builders zur Generierung der SQL-Abfragen bis hin zur Implementation einer abstrakten Abfragesprache. Die Möglichkeit der tranparenten Nutzung von Prepared Statements runden dies ab.

Zur Integration des Prototypen musste lediglich das *Install Tool* angepasst werden, während die Codebasis von TYPO3 CMS von den Umbaumaßnahmen nicht betroffen war. Aufgrund der gewählten Architektur des Prototypen kann TYPO3 CMS jedoch Schritt für Schritt auf die neue Datenbank API migriert werden.

AUSBLICK

Der implementierte Prototyp konnte im Rahmen des halbjährlichen Treffen, der TYPO3 CMS Kernentwickler präsentieren werden. Dabei wurde die Umsetzung durchweg positiv aufgenommen und eine mögliche Integration in TYPO3 CMS in Aussicht gestellt.[6] Mindestens zwei Kernentwickler haben dem Autor ihr Interesse an einer weiteren Zusammenarbeit ausgedrückt, bei der die Weiterentwicklung des Prototypen zu dem Nachfolger der Systemextension *DBAL* und der aktuellen Datenbank API im Fokus steht.

6 http://typo3.org/news/article/typo3-cms-active-contributors-meeting-2014/

[Ada95] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Englisch. Auflage: Reissue. Valley View, Calif.: Del Rey, Sep. 1995. isbn: 9780345391803.

[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.

[ISO92] ISOIEC. *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.

[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).

Adaptee

Eine Klasse die vone eine Adapterklasse in eine andere Klasse konvertiert wird.

SQL-Dialekt

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.

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

TABELLENVERZEICHNIS

Tabelle3_2

Alte API	Neue API
-	setDatabaseDriver()
-	getDatabaseDriver()
-	setConnectionParams()
-	getName()
-	getDatabaseCharset()
setDatabaseHost()	same
-	getDatabaseHost()
setDatabasePort()	same
-	getDatabasePort()
setDatabaseSocket()	same
-	getDatabaseSocket()
setDatabaseName()	same
-	getDatabaseName()
setDatabaseUsername()	same
-	getDatabaseUsername()
setDatabasePassword()	same
-	getDatabasePassword()
setConnectionCharset()	setDatabaseCharset()
-	getDatabaseCharset()
-	getDebugMode()
-	setDebugMode()
-	getPlatform()
-	getSchemaManager()
-	getSchema()
-	setConnected()
-	getLastStatement()
-	getStoreLastBuildQuery()
-	setStoreLastBuildQuery()
getDatabaseHandle()	same
setDatabaseHandle()	same
setPersistentDatabaseConnection()	same
setConnectionCompression()	same
setInitializeCommandsAfterConnect()	same
-	initCommandsAfterConnect()
setSqlMode()	same
initialize()	same
-	initDoctrine()
-	checkDatabasePreconditions()
-	prepareHooks()
connectDB()	connectDatabase()
-	close()
isConnected()	same
checkConnectionCharset()	same
disconnectIfConnected()	same
-	checkConnectivity()
query()	same
admin_query()	adminQuery()
sql_query()	adminQuery()
-	updateQuery()
-	createDeleteQuery()
exec_INSERTquery()	executeInsertQuery
exec_INSERTmultipleRows()	executeInsertMultipleRows
exec_UPDATEquery()	executeUpdateQuery
exec_DELETEquery()	executeDeleteQuery
exec_SELECTquery()	executeSelectQuery
exec_SELECT_mm_query()	executeSelectMmQuery
exec_SELECT_queryArray()	executeSelectQueryArray
exec_SELECTgetRows()	executeSelectGetRows

Seite 1

Tabelle3_2

exec_SELECTgetSingleRow()	executeSelectGetSingleRow
exec_SELECTcountRows()	executeSelectCountRows
exec_TRUNCATEquery()	executeTruncateQuery
INSERTquery()	createInsertQuery
INSERTmultipleRows()	createInsertMultipleRowsQuery
UPDATEquery()	createUpdateQuery
DELETEquery()	deleteQuery()
SELECTquery()	createSelectQuery
SELECTsubquery()	createSelectSubQuery
-	createTruncateQuery
TRUNCATEquery()	truncateQuery()
listQuery()	same
searchQuery()	same
prepare_SELECTquery()	prepareSelectQuery
prepare_SELECTqueryArray()	prepareSelectQueryArray
prepare_PREPAREDquery()	preparePreparedQuery
fullQuoteStr()	fullQuoteString
fullQuoteArray()	same
quoteStr()	quoteString
-	quoteIdentifier()
-	quoteTable()
-	quoteColumn()
-	quote()
escapeStrForLike()	escapeStringForLike
cleanIntArray()	cleanIntegerArray
cleanIntList()	cleanIntegerList
stripOrderBy()	same
stripGroupBy()	same
splitGroupOrderLimit()	same
getDateTimeFormats()	same
sql_error()	getErrorMessage()
sql_errno()	getErrorCode()
sql_num_rows()	getResultRowCount()
sql_fetch_assoc()	fetchAssoc()
sql_fetch_row()	fetchRow()
sql_free_result()	freeResult()
sql_insert_id()	getLastInsertId()
sql_affected_rows()	getAffectedRows()
sql_data_seek()	dataSeek()
sql_field_type()	getFieldType()
sql_pconnect()	getConnection()
sql_select_db()	selectDatabase()
admin_get_dbs()	listDatabases()
admin_get_tables()	listTables()
admin_get_fields()	listFields()
admin_get_keys()	listKeys()
admin_get_charsets()	listDatabaseCharsets()
handleDeprecatedConnectArguments()	same
debug()	same
debug_check_recordset()	debugCheckRecordset()
explain()	same
-	dropTable()
-	createTable()
-	countTables()
-	expr()
-	fetchColumn()

EIDESSTATTLICHE ERKLÄRUNG

Ich versichere, dass ich die vorliegende Thesis ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen benutzt habe.

Flensburg, den 26. Mai 2014

_____

Stefan Kowalke