MITTWALD
WEBHOSTING · SCHULUNGEN · SUPPORT

Entwicklerdokumentation

Extensionentwicklung ab TYPO3 4.3

Eine Einführung in Extbase und Fluid

Martin Helmich
Mittwald CM Service

Mittwald CM Service GmbH und Co. KG
Königsberger Straße 6
32339 Espelkamp

URL: http://www.mittwald.de
E-Mail: t3doku@mittwald.de

Inhalt

Die in der Dokumentation verwendeten Quelltexte sowie die erstellte Extension stehen

Abbildungen

Teil I: Einleitung

I.1 FLOW3 und Extbase

In der Vergangenheit wurden vorwiegend eher „kleine“ Anwendungen als TYPO3-Erweiterungen umgesetzt. In letzter Zeit lässt sich jedoch vor allem beobachten, dass TYPO3 zur Umsetzung zunehmend komplexerer Geschäftsapplikationen verwendet wird. Um der Rolle von TYPO3 als *Enterprise Content Management System* und den damit verbundenen immer weiter gestiegenen Anforderungen gerecht zu werden, stand bereits recht schnell nach der Gründung des Entwicklungsteams für TYPO3 5.0 fest, dass die nächste TYPO3-Version auf einem komplett neu entwickelten Gerüst aufsetzen sollte.

Dieses trägt den Namen FLOW3 und stellt als *Enterprise Application Framework* quasi ein „Programmiergerüst“ für darauf aufbauende Anwendungen dar. Neben TYPO3 kann FLOW3 also auch als Grundlage für beliebige andere – von TYPO3 unabhängige – Anwendungen verwendet werden.

Um den Übergang von dem aktuellen TYPO3-Zweig 4.x auf die 5er-Version zu erleichtern, stellte das TYPO3 5.0-Entwicklungsteam während der *TYPO3 Transition Days* im Oktober 2008 das sogenannte Berliner Manifest auf, welches klare Regeln für TYPO3 v4 und TYPO3 v5 aufstellt. So wurde zum Beispiel festgelegt, dass TYPO3 v4 auch nach dem Erscheinen von TYPO3 v5 weiterentwickelt wird, und dass Features, welche für TYPO3 v5 entwickelt werden, auch ihren Weg zurück in die Version 4 finden sollen.

So stehen zum Beispiel das MVC-Framework sowie die Templating-Engine Fluid – was genau das eigentlich ist, werden wir im später betrachten – bereits in TYPO3 4.3 in Form der Systemerweiterung „Extbase“ zur Verfügung. Diese ermöglicht es Entwicklern, bereits jetzt Erweiterungen zu entwickeln, die später ohne großen Aufwand nach TYPO3 v5 portiert werden können.

I.2 Zielgruppe und Voraussetzungen

Zielgruppe dieser Dokumentation sollen in erster Linie Entwickler sein, die bereits Kenntnisse über die Programmierung von TYPO3-Erweiterungen gesammelt haben.

Ziel dieser Dokumentation soll es nicht sein, einen kompletten Überblick über die Funktionsweise von TYPO3 zu bieten. Kenntnisse über die Grundlagen und Funktionsweise von TYPO3 werden daher an dieser Stelle als bekannt vorausgesetzt. Als Lektüre zum Einstieg sei an dieser Stelle auf die deutschsprachige TYPO3-Dokumentation[1] oder das Buch *Praxiswissen TYPO3* von Robert Meyer verwiesen.

Für die Entwicklung von TYPO3-Erweiterungen mit Extbase werden in dieser Dokumentation Grundkenntnisse in der Programmierung mit PHP vorausgesetzt. Hierzu sei zum Beispiel die offizielle PHP-Dokumentation zur Lektüre empfohlen.[2] Kenntnisse in der Erweiterungsentwicklung nach dem traditionellen Schema sind ebenfalls von Vorteil.

Kenntnisse über fortgeschrittene Entwicklungsmuster, wie zum Beispiel der objektorientierten Programmierung oder MVC-Architekturen, waren bisher bei der Entwicklung von TYPO3-Erweiterungen zwar von Vorteil, aber nicht zwingend vonnöten. Daher werden wir zu Beginn dieser Dokumentation zunächst noch einmal einen Blick auf diese theoretischen Hintergründe werfen und anschließend versuchen, diese Entwurfsmuster auf eine eigene Applikation anzuwenden. Als Beispiel soll hier eine Zeiterfassungsanwendung für Entwickler und Designer herhalten.

Anschließend werden wir im nächsten Kapitel in Form einer Schritt-für-Schritt-Anleitung eine erste Erweiterung auf Basis von Extbase entwickeln. Im Abschnitt „Vertiefung“ schließlich betrachten wir einige ausgewählte zusätzliche Aspekte, die für einfache Erweiterungen zwar nicht zwingend benötigt werden, sich in der Praxis jedoch als nützlich erwiesen haben.

1 http://www.mittwald.de/typo3-dokumentation/
2 http://www.php.net/manual/de/

Teil II: Design und Architektur

Am Anfang jedes Softwareprojekts steht ein durchdachter Entwurf der umzusetzenden Anwendung. Angesichts der zunehmenden Komplexität der Probleme in der Softwareentwicklung haben sich einige Konzepte bewährt, die dem Entwickler dabei helfen, auch in größeren Projekten „den Überblick“ zu bewahren.

Die bisherige Schnittstelle für die Entwicklung von TYPO3-Erweiterungen, die `pi_base`-Klasse, ließ dem Entwickler weitgehend freie Hand bei der Umsetzung seiner Projekte. Dem gegenüber versuchen sowohl Extbase als erst recht FLOW3, dem Programmierer einen etwas engeren Rahmen zu setzen. Beide sind speziell auf die Softwareentwicklung nach bestimmten Designparadigmen, dem *Domain-Driven Design* und der *Model-View-Controller*-Architektur ausgelegt.

Diese Gegebenheit mag vielen Extension-Entwicklern zunächst als Einschränkung erscheinen. Und tatsächlich bringt der Umstieg auf Extbase – und später auf FLOW3 – einige hohe Einstiegshürden mit sich, wird nun doch schließlich ein gewisses Vorwissen über fortgeschrittene Softwareentwurfs- und Architekturmuster vorausgesetzt. Andererseits ermöglichen gerade diese Muster dem Programmierer, seinen Code klar, übersichtlich und modular zu gestalten. Dies wiederum vereinfacht die spätere Umsetzung und die Wartung der Software ungemein. Nicht zuletzt hierdurch entsteht auch ein enormes Potential zum Einsparen von Entwicklungskosten.

Bevor wir also mit der tatsächlichen Umsetzung unserer ersten eigenen Erweiterung mit Extbase beginnen, halten wir einen Moment inne, um einen Entwurf anzufertigen. Dabei werden wir auf einige in der Softwareentwicklung etablierte Design-Methoden und Konzepte zurückgreifen, welche im ersten Teil dieses Kapitels zunächst theoretisch erläutert werden sollen. Anschließend werden wir versuchen, die erlernten Konzepte auf unser eigenes Projekt anzuwenden.

II.1 Grundlegende Konzepte

Im folgenden Kapitel werden wir zunächst einige Konzepte betrachten, die der Entwicklung mit Extbase und FLOW3 zugrunde liegen. Dabei werden wir zuerst einen kurzen Blick auf die Objektorientierte Programmierung werfen, anschließend auf das *Domain-Driven Design* und letztendlich auf die *Model-View-Controller*-Architektur.

Zweifellos werden einige dieser Konzepte vielen oder sogar den meisten Lesern bereits geläufig sein. So hat zum Beispiel die objektorientierte Programmierung bereits seit vielen Jahren einen festen Platz in den Lehrplänen der meisten Lehrer und Professoren. Auch das MVC-Muster wird dem einen oder anderen Webentwickler bereits über den Weg gelaufen sein. Dennoch war bisher bei der Entwicklung von Erweiterungen für TYPO3 kein Vorwissen über die meisten dieser Konzepte notwendig, weshalb wir an dieser Stelle zumindest einmal einen kurzen Blick darauf werfen möchten.

In diesem Kapitel werden wir noch nicht direkt mit Extbase arbeiten. Wenn Sie sich also bisher noch nicht mit Extbase beschäftigt haben, aber zum Beispiel mit der objektorientierten Programmierung bereits vertraut sind, so können Sie die entsprechenden Kapitel ruhigen Gewissens überspringen.

Zu beachten ist, dass über jeden dieser Punkte bereits ganze Bücher geschrieben wurden. Ich werde mir daher erlauben, im folgenden Kapitel nur einen kurzen Überblick über die entsprechenden Themen zu bieten und zur Vertiefung des jeweiligen Themas auf weiterführende Literatur zu verweisen.

II.1.1 Objektorientierte Programmierung

II.1.1.1 Überblick

Das Konzept der *objektorientierten Programmierung* wurde bereits in den 1960er Jahren entwickelt und gehört seit Anfang der 1990er Jahre zu einem der Grundkonzepte der Softwareentwicklung. Während viele höhere Programmiersprachen, wie C++ und Java, das Konzept bereits frühzeitig vollständig implementierten, wurde das Thema in PHP lange Zeit eher stiefmütterlich behandelt. Eine umfassendes Modell zum objektorientierten Programmieren steht erst seit dem 2004 erschienenen PHP 5 zur Verfügung.

Ziele der Objektorientierung sind die klarere Strukturierung und einfachere Wiederverwendbarkeit von Programmquelltexten, und somit eine Reduzierung des Entwicklungsaufwandes.

II.1.1.2 Einführung

II.1.1.2.a Objekte und Klassen

Die grundlegende Idee hinter der Objektorientierung ist es, gleichartige Daten in einem sogenannten *Objekt* zusammen zu fassen. Solche Objekte sind nach außen hin *gekapselt*, sodass auf die Daten in dem Objekt nur über wohldefinierte Schnittstellen zugegriffen werden kann.

Um gleichartige Objekte besser verwalten zu können, werden sogenannte *Klassen* als Vorlage für solche Objekte verwendet. Ein Objekt, welches aus solch einer Klasse erstellt wird, heißt dann *Instanz* dieser Klasse. Jedes Objekt hat bestimmte Eigenschaften, die in der Fachsprache *Attribute* genannt werden. Außerdem können mit einem Objekt Aktionen durchgeführt werden, die sogenannten *Methoden*.

Eines der klassischen Lehrbuch-Beispiele wäre eine Klasse mit dem Namen *Auto*. Ein Auto zeichnet sich durch diverse Eigenschaften aus, wie zum Beispiel der Marke, Farbe, Alter, Kilometerstand, Tankfüllung und vielem mehr. Gleichzeitig können Aktionen mit dem Auto durchgeführt werden: wir können damit fahren, es auftanken, umlackieren, und so weiter. Die Klasse *Auto* ist jedoch nur eine abstrakte Vorlage für bestimmte Objekte. Eine Instanz der Klasse *Auto* könnte also ein ganz bestimmtes Auto sein, also zum Beispiel ein weißer VW Golf, Baujahr 1999, mit einem Kilometerstand von 150.000 Kilometern und einem Tankfüllstand von 15 Litern.

II.1.1.2.b Datenkapselung

Ein weiteres Prinzip der objektorientierten Programmierung ist es, dass auf die Daten innerhalb eines Objektes nicht direkt zugegriffen werden kann. Stattdessen muss das Objekt entsprechende Schnittstellen für die Außenwelt bereitstellen. Dies sind in der Regel spezielle Methoden, sogenannte *Zugriffsfunktionen*.

Methoden, die zum Auslesen von Daten dienen, werden dabei häufig als *sondierende Methoden* (engl.: *getter methods*) bezeichnet, während Methoden, die Daten verändern, als *manipulierende Methoden* (oder engl. *setter methods*) bezeichnet werden. Analog dazu hat es sich als Konvention eingebürgert, sondierende Methoden mit dem Präfix *get* zu benennen (also zum Beispiel *getColor* beim Auto) und manipulierende Methoden mit *set* zu benennen (*setColor*).

II.1.1.2.c Beispiel: Objektorientierung in PHP

In PHP wird eine Klasse mit dem *class*-Statement deklariert. Innerhalb der Klasse folgen – durch geschweifte Klammern abgetrennt – zunächst die Attribute, optional mit Zugriffsangabe, anschließend die Methoden. Auf die verschiedenen Zugriffsangaben wird im nächsten Abschnitt eingegangen.

```
class Auto {
    private $color;
    private $mileage;
    private $built;
    private $brand;

    public function setColor($color) {
        $this->color = $color;
    }

    public function setMileage($mileage) {
        $this->mileage = $mileage;
    }

    public function setBuildYear($built) {
        $this->build = $built;
    }

    public function setBrand($brand) {
        $this->brand = $brand;
    }

    public function print() {
        echo "Marke: {$this->brand}, Farbe: {$this->color}, ".
             "Baujahr: {$this->built}\n";
    }
}
```

Das Schlüsselwort `$this` erlaubt einen Zugriff auf die Attribute und Methoden des jeweils aktuellen Objektes.

Eine Instanz dieser Klasse kann dann schließlich mit dem *new*-Schlüsselwort erstellt werden:

```
$auto1 = new Auto();
$auto2 = new Auto();

$auto1->setBrand("Volkswagen");
$auto1->setBuildYear(2007);
$auto1->setColor("Blau");
```

```
$auto2->setBrand("Toyota");
$auto2->setBuildYear(2010);
$auto2->setColor("Weiß");

$auto1->print();
$auto2->print();

// Ausgabe:
// Marke: Volkswagen, Farbe: Blau, Baujahr: 2007
// Marke: Toyota, Farbe: Weiß, Baujahr: 2010
```

II.1.1.3 Vertiefung

II.1.1.3.a Sichtbarkeit von Attributen und Methoden

Für einzelne Attribute und Methoden können verschiedene *Sichtbarkeiten* festgelegt werden. Der UML-Standard kennt hier drei verschiedene Stufen (genau genommen sind es sogar vier; PHP kennt jedoch nur drei davon, daher reichen uns diese hier):

1. *Öffentlich*. Öffentliche Attribute und Methoden sind nach außen hin sichtbar. Dies bedeutet, dass auch andere Objekte auf diese Attribute und Methoden zugreifen dürfen. In UML-Notation werden öffentliche Eigenschaften mit einem + gekennzeichnet.
2. *Geschützt*. Auf Attribute und Methoden, die als geschützt markiert, kann nicht von außen zugegriffen werden. Ein Zugriff ist nur aus derselben Klasse oder aus abgeleiteten Klassen möglich. UML-Notation für geschützte Attribute ist das #-Zeichen.
3. *Privat*. Genau wie auf geschützte Attribute und Methoden kann auf private Eigenschaften ebenfalls nicht von außen zugegriffen werden. Im Unterschied zu geschützten Eigenschaften aber kann auf private Attribute und Methoden nicht aus einer Unterklasse heraus zugegriffen werden. In der UML werden private Eigenschaften mit einem „-“-Zeichen gekennzeichnet.

PHP kennt zur Deklaration der Sichtbarkeit die Schlüsselwörter *public*, *protected* und *private*:

```
Class Test {
    Public $foo = 'Hallo';
    Protected $bar = 'Welt';
    Private $baz = 'Und tschüss!';
}
```

```
$test = New Test();
echo $test->foo;   // Erlaubt
echo $test->bar;   // Nicht erlaubt
echo $test->baz;   // Nicht erlaubt
```

II.1.1.3.b Statische Methoden und Attribute

Statische Methoden und Attribute sind zwar einer Klasse zugeordnet, können aber aufgerufen werden, ohne dass vorher eine Instanz dieser Klasse erstellt werden muss. Aus einer statischen Methode kann auch auf geschützte oder private Attribute und Methoden innerhalb einer Instanz derselben Klasse zugegriffen werden.

```
Class Test {
    Static Public $foo = "bar";

    Static Public Function test() { echo "Hallo Welt!"; }
}

echo Test::$foo;   // Ausgabe: bar
Test::test();      // Ausgabe: Hallo Welt!
```

II.1.1.3.c Vererbung

Eine Klasse kann Methoden und Attribute von einer anderen Klasse „erben“. Hierzu wird in PHP das Schlüsselwort *extends* verwendet. Auf diese Weise stehen alle Attribute und Methoden der übergeordneten Klasse auch in der untergeordneten Klasse zur Verfügung. Diese kann dann nach Belieben um zusätzliche Methoden und Attribute ergänzt werden.

Geerbte Methoden und Attribute können überschrieben werden, indem sie einfach unter demselben Namen neu deklariert werden. Dies ist nur möglich, wenn die entsprechende Methode in der Elternklasse nicht mit dem Schlüsselwort *final* geschützt wurde.

```
Class Auto {
    Public Function losfahren() { echo "Töff, töff...\n"; }
    Public Function bremsen() { echo "Quieetsch!\n"; }
}

Class SchnellesAuto Extends Auto {
    Public Function losfahren() { echo "Wrrrummm...\n"; }
}

$auto = New Auto();
$schnellesAuto = New SchnellesAuto();
```

```
$auto->losfahren();  // Töff, töff...
$auto->bremsen();    // Quieetsch!

$schnellesAuto->losfahren();  // Wrrrummm...
$schnellesAuto->bremsen();    // Quieetsch!
```

II.1.1.3.d Abstrakte Klassen

Eine abstrakte Klasse ist eine Klasse, von der keine Instanz erstellt werden kann. Sie existiert lediglich zu dem Zweck, damit andere Klassen Attribute und Methoden von ihr erben können. Diese Klassen heißen dann *konkrete Klassen*. In PHP werden abstrakte Klassen durch das Schlüsselwort *abstract* markiert.

Abstrakte Klassen dürfen wiederum *abstrakte Methoden* enthalten. Diese Methoden haben enthalten keine Programmlogik; diese muss dann zwingend von den erbenden Unterklassen implementiert werden.

```
Abstract Class Lebewesen {
    Abstract Public Function geraeuschMachen();
}

Class Hund {
    Public Function geraeuschMachen() { echo "Wau!"; }
}

Class Katze {
    Public Function geraeuschMachen() { echo "Miau!"; }
}

$lebewesen = New Lebewesen();
$lebewesen->geraeuschMachen();  // Geht nicht!

$hund = New Hund();
$hund->geraeuschMachen();       // Wau!

$katze = New Katze();
$katze->geraeuschMachen();      // Miau!
```

II.1.1.3.e Schnittstellen

Eine Schnittstelle (engl. *interface*) definiert einen Satz von öffentlichen Methoden, die eine Klasse bereitstellen muss. Eine solche Schnittstelle kann von beliebig vielen Klassen implementiert werden. Im Gegensatz zur Vererbung kann allerdings jede Klasse beliebig viele Schnittstellen implementieren.

Anders als eine abstrakte Klasse darf eine Schnittstelle jedoch keine Programmlogik enthalten.

```
Interface Auftankbar {
    Public Function auftanken();
}

Abstract Class Fortbewegungsmittel {
    Public Function fahre($km) {
        echo "Fahre $km Kilometer mit dem ".get_class($this);
    }
}

Class Fahrrad Extends Fortbewegungsmittel {  }

Class Auto Extends Fortbewegungsmittel Implements Auftankbar {
    Public Function auftanken() {
        echo "Auto mit 50 Litern Benzin aufgetankt."
    }
}

Class Schiff Extends Fortbewegungsmittel Implements Auftankbar {
    Public Function auftanken() {
        echo "Schiff mit 1000 Litern Diesel aufgetankt."
    }
}

$fahrrad = New Fahrrad();
$fahrrad->fahre(20);       // Fahre 20 Kilometer mit dem Fahrrad
$fahrrad->auftanken();     // Geht nicht!

$auto = New Auto();
$auto->fahre(100);         // Fahre 100 Kilometer mit dem Auto
$auto->auftanken();        // Auto mit 50 Litern Benzin aufgetankt

$schiff = New Schiff();
$schiff->fahre(2000);      // Fahre 2000 Kilometer mit dem Schiff
$schiff->auftanken();      // Schiff mit 1000 Litern Diesel aufgetankt
```

II.1.2 Domain-Driven Design

II.1.2.1 Grundlagen

Das Domain-Driven Design ist ein Begriff, der 2003 von Eric Evans in seinem Buch *Domain-Driven Design: Tackling Complexity in the Heart of Software* geprägt wurde. Dabei handelt es sich um eine Herangehensweise für den Entwurf komplexer, objektorientierter Software. Vorweg gilt es klarzustellen, dass es sich beim Domain-Driven Design nicht um ein konkretes Architekturmuster handelt, sondern vielmehr um eine Denkweise zum Entwurf von Software.

II.1.2.1.a Domänen und Fachmodelle

Im Zentrum des *Domain-Driven Designs* steht grundsätzlich eine sogenannte *Domäne*. Darunter versteht man ein bestimmtes Problemfeld der realen Welt, deren Strukturen und Prozesse in der zu entwickelnden Software abgebildet werden sollen. Die Domäne eines Informationssystems für Hotels sollte zum Beispiel sowohl die Hotelgäste, die zur Verfügung stehenden Zimmer sowie alle verbundenen Prozesse, wie zum Beispiel das Ein- und Auschecken abbilden können.

Da es nicht möglich ist, die reale Welt in hundertprozentiger Genauigkeit abzubilden, benötigen wir zunächst ein Modell, welches die reale Welt vereinfacht und abstrakt abbildet. Jede Domäne basiert also auf einem solchen *Fach-* oder *Domänenmodell*, welches nur die für diese Domäne interessanten Aspekte darstellt. So interessiert uns in obigem Szenario etwa nicht die Farbe der Vorhänge in einem der Hotelzimmer, dafür aber eventuell die Anzahl der Betten, Größe, Preis oder Buchungen für dieses Zimmer.

Wie wir also sehen, ist es weder das Ziel des Domain-Driven Designs, die technische Umsetzung der Anwendung festzulegen, noch eine einhundert-prozentige Abbildung der Wirklichkeit zu schaffen. Stattdessen sollen vielmehr Klarheit über die Prozesse und die Funktionsweise Ihrer Anwendung zu geschaffen, sowie ein Grundgerüst für den Aufbau der späteren Anwendung hergestellt werden.

II.1.2.1.b Kommunikation

Die Anwendung des *Domain-Driven Designs* setzt nach Evans voraus, dass der Entwicklungsprozess *iterativ* erfolgt. Dies ist eine Methode, die vor allem in der Agilen Softwareentwicklung Anwendung findet. Das bedeutet, dass die Entwicklung in mehreren Durchläufen (Iterationen) erfolgt, in welchen bei jedem Durchlauf eine erneute Evaluierung und Analyse erfolgt. Vorteile dieses Ansatzes liegen darin, dass sich die Ent-

wickler das Wissen zunutze machen können, welches erst im Verlauf der Entwicklung gewonnen wird. Des weiteren können auch während der Entwicklung wiederholt die Anforderungen überprüft und gegebenenfalls angepasst werden. All dies ermöglicht eine höhere Flexibilität bei der Entwicklung, erfordert aber eine ständige Kommunikation und enge Zusammenarbeit zwischen allen Beteiligten.

Beteiligte sind klassischerweise der Auftraggeber und der Auftragsnehmer. Der Auftraggeber ist Experte im Umfeld der Anwendungsdomäne (daher nach Evans die Bezeichnung *domain expert* oder *Fachexperte*), während der Auftragsnehmer oder Entwickler diese Domäne schließlich in der Software abbilden muss. Aus diesem Grund ist eine ausführliche Kommunikation zwischen diesen beiden Beteiligten unerlässlich.

Da diese häufig aus unterschiedlichen „Welten“ stammen, sind Kommunikationsprobleme nicht selten bereits vorprogrammiert. Um diese zu vermeiden, schlägt Evans die Verwendung einer sogenannten *ubiquitären Sprache* vor. Hinter diesem zugegebenermaßen schon nahezu unheimlich klingenden Wortungetüm verbirgt sich die Idee, dass sich alle Beteiligten auf eine gemeinsames Vokabular einigen, welches um das Domänenmodell herum strukturiert ist, und das sowohl von Entwicklern als auch Fachexperten verstanden wird. Diese Sprache sollte konsequent verwendet werden, sowohl bei der Benennung von Programmkomponenten (also zum Beispiel Klassen, etc.), als auch in der Dokumentation und in jeder Form von Kommunikation.

Bei der Erstellung der gemeinsamen Sprache sollten also sowohl die Entwickler auf die Verwendung allzu technischer Begriffe, zum Beispiel in Richtung Datenbankentwurf, oder bestimmte Architekturmuster, verzichten, als auch die Fachexperten auf die Verwendung zu spezieller Fachbegriffe. Auf diese Weise können lästige und zeitraubende Missverständnisse von Anfang an vermieden werden.

II.1.2.2 Schichtenarchitektur

Das Domain-Driven Design setzt die Verwendung einer *Schichtenarchitektur* voraus. Evans benennt hierfür die folgenden Schichten:

1. **Benutzerschnittstelle.** Die Benutzerschnittstelle präsentiert die Anwendung dem Benutzer und nimmt Eingaben entgegen.
2. **Anwendung.** Diese Schicht nimmt Benutzereingaben entgegen und koordiniert die Zusammenarbeit mit dem Domänenmodell in der nächsten Schicht. Die Anwendungsschicht selbst enthält keine Geschäftslogik – dies ist die Aufgabe der Domäne.
3. **Domäne.** Das Domänenmodell ist das „Herz“ der Anwendung. Es bildet eine Domäne der realen Welt ab und enthält die komplette Geschäftslogik der Anwendung. Technische Details der Domäne werden an die Infrastrukturschicht weitergereicht.
4. **Infrastruktur.** Diese Schicht stellt die technische Infrastruktur für die Anwendung bereit. Dies beinhaltet zum Beispiel die dauerhafte Speicherung von Objekten in einer Datenbank.

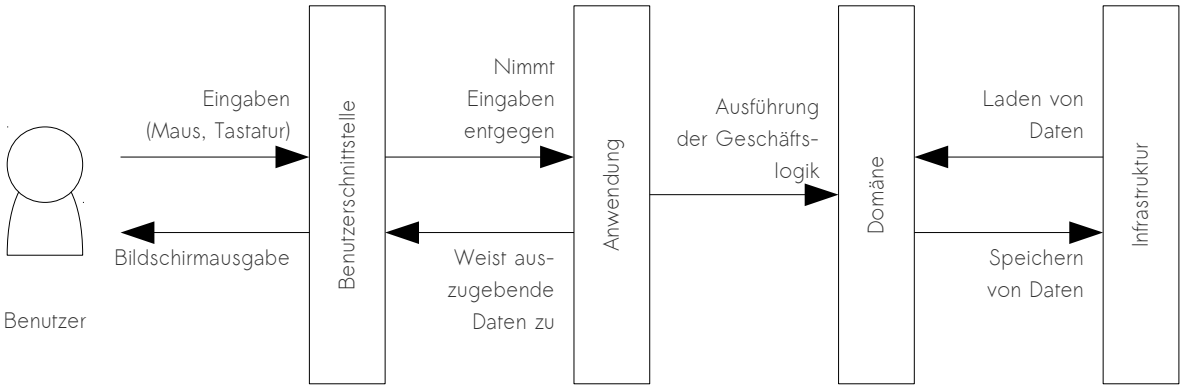

Abbildung 1: Beispiel für eine Schichtenarchitektur

Der Fokus des Domain-Driven Design liegt nun auf der Domänenschicht und dem Fachmodell. Durch die Aufteilung in mehrere Schichten bleiben die Domänenobjekte selbst frei von „lästigen“ alltäglichen Aufgaben, wie der Präsentation, Validierung von Benutzereingaben, Suche von Objekten oder der Notwendigkeit, Daten in einer Datenbank speichern zu müssen.

Durch die klare Trennung der Zuständigkeiten können die Entwickler sich zunächst auf das Wesentliche konzentrieren, nämlich die Anwendungsdomäne. Nicht zuletzt hierdurch erhöht sich die Wiederverwendbarkeit und auch die Übersichtlichkeit des Quelltextes.

Die hier besprochene Schichtenarchitektur werden wir später im Rahmen der *Model-View-Controller*-Architektur noch einmal wieder aufgreifen.

II.1.2.3 Das Fachmodell

Das Domänenmodell besteht aus Objekten verschiedener Typen. Wie wir es von objektorientierten Ansatz her kennen, hat jedes Objekt eine definierte Menge an Eigenschaften, oder Attributen, sowie einen bestimmten Satz Methoden.

Nach dem Ansatz von Evans werden diese Objekte nun noch weiter in *Entitäten* und *Wertobjekte* (engl. *entities* und *value objects*) unterteilt. Diese Objekte können in verschiedenartigen Beziehungen zueinander stehen, welche *Assoziationen* genannt werden. Weitere Bestandteile des Modells sind *Dienste* (engl. *services*) sowie *Domänenereignisse* (engl. *domain events*). Sämtliche dieser Bestandteile werden im folgenden Abschnitt näher erläutert.

II.1.2.3.a Entitäten und Wertobjekte

Wie bereits erläutert, hat jedes Objekt eine bestimmte Menge an Attributen. Eine *Entität* ist nun ein Objekt, welches sich nicht allein über seine Attribute eindeutig definieren lässt. Ein einfaches Beispiel für solch eine Entität wäre eine *Person*, die das Attribut *Name* trägt. Nun kann es durchaus vorkommen, dass zwei Personen denselben Namen tragen; dennoch handelt es sich um zwei verschiedene Personen. Anders herum kann eine Person ihren Namen ändern; bleibt aber dennoch dieselbe Person.

Aus diesem Grund werden für Entitäten meistens zusätzliche, eindeutige Identifikatoren verwendet. Im TYPO3-Umfeld ist dies meist der *unique identifier* – die UID, ein numerischer Index, der für Datenbankzeilen vergeben wird. FLOW3 hingegen verwendet einen 128-Bit-Hexadezimalzahl als Kennzeichner. Tatsächlich ist die Implementierung eines solchen Identifikators dem Entwickler überlassen, solange die Eindeutigkeit gewährleistet ist.

Im Gegensatz zur Entität kann ein *Wertobjekt* eindeutig über seine Attribute definiert werden. Ein Beispiel für ein solches Wertobjekt wäre etwa eine Adresse, charakterisiert beispielsweise durch die Attribute Straße, Hausnummer und PLZ/Ort. Stimmen nun zwei Adressen in all diesen Attributen überein, handelt es sich zweifellos um dieselbe Adresse. Verändern wir allerdings die Postleitzahl eines Adress-Objektes, so beschreibt das Objekt danach – logischerweise – eine andere Adresse als vorher.

II.1.2.3.b Assoziationen

Zwei oder mehrere Domänenobjekte können in Beziehung zueinander stehen. So können einem Kunden beispielsweise etwa mehrere Aufträge zugeordnet werden. Die Definition solcher Assoziationen entspricht im großen und ganzen der des UML-Standards. Demnach kann eine Assoziation unter anderem durch ihre *Multiplizität* und ihre *Navigierbarkeit* charakterisiert werden.

- **Multiplizität.** Die Multiplizität definiert, wie viele Objekte zueinander in Beziehung stehen dürfen. Häufig wird sie als Intervall beschrieben, innerhalb dessen die Anzahl der assoziierten Objekte liegen darf. Gebräuchliche Multiplizitäten sind zum Beispiel *0..1* (höchstens ein assoziiertes Objekt), 1 (genau ein assoziiertes Objekt) oder *0..** (beliebig viele assoziierte Objekte). Die UML-Notation einer solchen Assoziation sähe beispielsweise aus wie folgt:

Abbildung 2: Assoziation zwischen Objekten

  In diesem Beispiel könnten einem einzelnen Kunden also 0 bis beliebig viele Aufträge zugeordnet werden; jedem einzelnen Auftrag allerdings nur genau ein einzelner Kunde.

- **Navigierbarkeit.** Die Navigierbarkeit beschreibt, in welche Richtung eine solche Assoziation abgefragt werden darf. Eine Assoziation, die in beide Richtungen navigiert werden darf, heißt *bidirektionale Assoziation*. Im obigen Beispiel müsste die Klasse *Kunde* dafür eine Methode *gibAuftraege()*, und die Klasse *Auftrag* eine Methode *gibKunde()* implementieren. Kann die Assoziation nur in einer Richtung navigiert werden, spricht man von einer *unidirektionalen Assoziation*.

Eine besondere Art der Assoziationen ist die *Komposition*. Bei einer Komposition handelt es sich um eine *1-zu-0..**-Beziehung zwischen zwei Objekten. Ein Beispiel wäre etwa die bereits oben dargestellte Beziehung zwischen Kunden und Aufträgen: Jeder Kunde kann beliebig viele Aufträge erteilen, jeder Auftrag kann aber nur einem ganz bestimmten Kunden zugeordnet werden. Wenn ein Objekt einer Komposition gelöscht wird, macht es in der Regel Sinn, auch die assoziierten Objekte zu löschen; wenn der Kunde gelöscht ist, können auch die Aufträge gelöscht werden.

II.1.2.3.c Dienste und Module

Weitere Bestandteile des Domänenmodells können Dienste, Ereignisse und Module sein. Da wir diese beim Arbeiten mit Extbase und FLOW3 nur eingeschränkt verwenden, seien diese an hier nur der Vollständigkeit halber erwähnt.

Dienste (engl. *services*) beinhalten Funktionalitäten, die konzeptionell keinem einzelnen Objekt der Domäne zugeordnet werden können. Diese werden in einem einzelnen Serviceobjekt, welches keine weiteren Assoziationen hat, ausgelagert. Zwar gibt es in Extbase keine eigene Schnittstelle für solche Dienste, dennoch ist eine eigene Implementation selbstverständlich möglich. In FLOW3 bietet das AOP-Framework (Aspektorientierte Programmierung) ähnliche Funktionalitäten.

Module teilen die Domäne in fachliche Untergruppen. Per Definition zeichnet sich ein Modul durch starke *innere Kohäsion* sowie durch *geringe Kopplung* zu anderen Modulen aus. Dies bedeutet, dass einerseits in einem Modul Objekte zusammengefasst werden sollten, die auch wirklich eng zueinander gehören, und andererseits die Anzahl der Schnittstellen zwischen verschiedenen Modulen überschaubar bleiben sollte.

Auch die Modularisierung ist letzten Endes eine Form der Kommunikation. Dadurch, dass mehrere Objekte in einem Modul zusammengefasst werden, wird dem nächsten Entwickler, der an dem Projekt arbeitet, mitgeteilt, dass diese Objekte logisch zusammen gehören.

II.1.2.4 Lebenszyklus von Objekten

Als Lebenszyklus (engl. *life cycle*) werden die Zustände bezeichnet, die ein Objekt im Verlauf eines Programmablaufs durchläuft. Bei den meisten Objekten ist dies recht einfach; sie werden erstellt, einige ihrer Methoden werden aufgerufen und dann werden sie wieder gelöscht. Bei einigen Objekten verhält es sich jedoch komplizierter; so sollen die meisten Domänenobjekte schließlich auch beim nächsten Programmdurchlauf noch vorhanden sein. Diese Objekte müssen also *persistent* (dauerhaft) gespeichert und wieder

geladen werden können. Noch komplizierter wird es, wenn ein Objekt eine Assoziation zu einem oder mehreren weiteren persistenten Objekten hat. In diesem Fall ergeben sich weitere Probleme, zum Beispiel zu welchem Zeitpunkt assoziierte Objekte geladen oder wieder gespeichert werden sollen.

Solche Objekte werden klassischerweise entweder serialisiert oder über eine objektrelationale Abbildung in einer Datenbank gespeichert. Tatsächlich hat es uns beim Arbeiten mit einem Framework wie FLOW3 oder auch Extbase nicht zu kümmern, wie unsere Daten dauerhaft gespeichert werden – dafür haben wir schließlich das Framework. Zumindest bei Extbase ist dies jedoch nur die halbe Wahrheit – um ein wenig Datenbankdesign kommen wir leider nicht herum; doch dazu später mehr.

Wie bereits mehrfach erwähnt handelt es sich beim Domain-Driven Design um ein Entwurfsmuster, und nicht um ein Konzept zur technischen Umsetzung. Wie genau die Domänenobjekte persistent gespeichert werden, bleibt an dieser Stelle also zunächst offen. Das Entwurfskonzept bringt jedoch einen Ansatz mit sich, wie der Zugriff auf solche Objekte geregelt werden soll. In diesem Rahmen werden wir im folgenden Abschnitt zunächst einen genaueren Blick auf *Aggregatstrukturen* werden, sowie anschließend auf *Repositories* und *Factories*.

II.1.2.4.a Aggregate

Im vorigen Abschnitt hatten wir bereits die Komposition erwähnt. In diesem Fall stehen zwei Objekte in einer derartigen Beziehung zueinander, dass ein Objekt einem anderen „gehört“ – zum Beispiel ein Auftrag einem Kunden. Solche Kompositionen können beliebig verzweigt werden. So könnten einem Auftrag zum Beispiel wieder beliebig viele Auftragspositionen zugeordnet werden. Des weiterem könnten jedem Kunden zusätzlich zu seinen Aufträgen auch mehrere Lieferadressen zugeordnet werden:

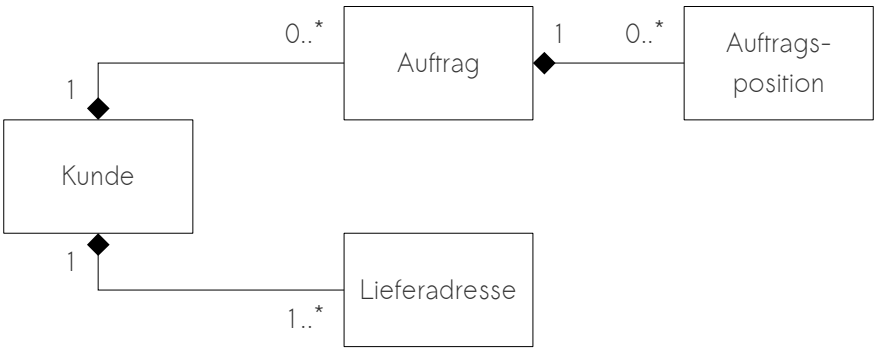

Abbildung 3: Verschiedene Kompositionen im Domänenmodell

Auf den ersten Blick ist offensichtlich, dass alle beteiligten Objekte direkt oder indirekt vom Kunden abhängig sind. Wenn der Kunde aus irgendeinem Grund gelöscht wird, müssen auch alle Lieferadressen sowie alle Aufträge, und damit auch alle Auftragspositionen, gelöscht werden.[3]

Eine solche Gruppe von Objekten, die als eine einzelne Einheit betrachtet werden, wird als *Aggregat* bezeichnet.[4] Jedes Aggregat zeichnet sich durch eine *Wurzel* (engl. *aggregate root*) sowie eine Begrenzung aus, die definiert, welche Objekte Teil des Aggregats sind, und welche nicht. Die Aggregatwurzel ist dabei eine Entität (Definition siehe oben) und gleichzeitig der einzige Bestandteil des Aggregates, auf den von Objekten außerhalb des Aggregates zugegriffen werden kann. Des weiteren ist die Aggregatwurzel das einzige Objekt, welches *globale Identität* besitzen muss. Alle anderen Objekte müssen lediglich innerhalb des Aggregates eindeutig identifizierbar sein; von außerhalb kann ja ohnehin nicht darauf zugegriffen werden.

In unserem Beispiel würden die Klassen *Kunde* und *Lieferadresse* ein Aggregat bilden. Die Kundenentität dient dabei als Aggregatwurzel. Ein direkter Zugriff auf die Lieferadresse ist hier nicht möglich. Tatsächlich erscheint es auch eher unrealistisch, dass in einem Warenwirtschaftssystem anhand einer Lieferadresse der dazugehörige Kunden ermittelt werden soll. Wahrscheinlicher wird andersherum nach dem Kunden gesucht, um die Lieferadresse zu finden.

Schon realistischer ist jedoch, dass jemand direkt nach einer Auftragsnummer sucht, um den dazugehörigen Kunden zu ermitteln. Das Objekt *Auftrag* wäre nach dieser Definition also kein Bestandteil des Aggregates, da auch ein direkter Zugriff darauf möglich sein muss. Jedoch könnte auch der *Auftrag* wieder als Aggregatwurzel für die *Auftragspositionen* betrachtet werden:

3 In der Realität würde natürlich niemand einfach einen Kunden mit allen Aufträgen aus einem Warenwirtschaftssystem löschen. Hier geht es nur um die Veranschaulichung des Prinzips.

4 Der UML-Standard kennt auch den Begriff der *Aggregation*. Trotz des ähnlichen Wortlautes geht es hier um zwei unterschiedliche Dinge.

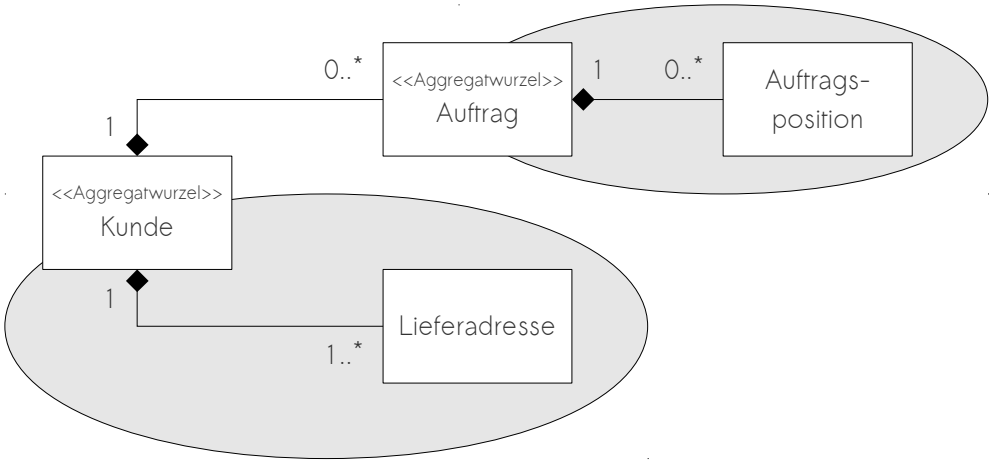

Abbildung 4: Aggregate im Domänenmodell

Fassen wir das Konzept noch einmal zusammen: Auf alle Objekte, die untergeordneter Bestandteil eines Aggregates sind, darf ausschließlich über die Aggretatwurzel zugegriffen werden. Haben wir also eine Klasse *Kunde*, so muss diese eine Methode *gibLieferadressen()* implementieren. Diese Zugriffsmethode wird als *Zugriff durch Traversion* bezeichnet (engl. *access by traversal*). Welche Möglichkeit aber haben wir, auf die Aggregatwurzeln zuzugreifen? An dieser Stelle kommen die *Repositories* ins Spiel.

II.1.2.4.b Repositories

Nun gibt es einige Entitäten, auf die kein oder nur ein unpraktischer Zugriff durch Traversion möglich ist. Üblicherweise sind dies Wurzelobjekte von Aggregaten. Um einen globalen Zugriff auf solche Objekte zu ermöglichen, wird das Domänenmodell für jedes dieser Fachobjekte um ein sogenanntes *Repository* erweitert (dt. in etwa *Aufbewahrungs-* oder *Verwahrungsort*; da die Übersetzung jedoch ein wenig holpert, verwenden wir im Folgenden den englischsprachigen Begriff).

Ein Repository gehört zwar genau genommen nicht zum Fachmodell, wird aber allgemein dennoch zur Domäne dazu gezählt, da es für den Zugriff auf Objekte wichtige Funktionen bereitstellt. Wichtigste Aufgabe eines Repositories ist es, eine Abstraktionsschicht zwischen der Anwendungsdomäne und der Infrastruktur – beispielsweise einem relationalen Datenbankmanagementsystem – darzustellen.

Auf diese Weise können andere Objekte der Domäne oder die Anwendungsschicht beim Repository ein bestimmtes Objekt anhand festgelegter Suchkriterien anfragen („Gib mir den Kunden mit der Kundennummer XYZ“, „Gib mir alle Kunden aus Deutschland“). So bleibt die Domäne frei von lästigen Zugriffen auf die Infrastruktur. Des weiteren braucht es die Anwendung nicht zu kümmern, woher das Repository seine Daten eigentlich bezieht; ob nun direkt aus einem RDBMS, einem vorgelagerten Cache oder sogar per Webservice von einem anderen Server.

II.1.2.4.c Fabriken

Eine Fabrik (engl. *factory*) wird für die Erstellung komplexer Fachobjekte benötigt. Gerade in komplexen Domänen wird allein für das Anlegen neuer Objekte eine große Menge Programmlogik verwendet. Da diese im Lebenszyklus des Objekts nur einmal benötigt wird – und zwar beim Erstellen – macht es unter Umständen Sinn, diese Logik in ein eigenes Objekt auszulagern: eine Fabrik.

Ein Beispiel, in dem die Verwendung einer Fabrik Sinn machen könnte, wäre etwa das Neu-Anlegen eines Kundenobjektes. Komplizierte Mechanismen wie das Erstellen einer neuen Kundennummer werden ausschließlich dann benötigt, wenn ein neuer Kunde erstellt wird. Aus diesem Grund macht es Sinn, derartige Methoden aus dem Kunden-Objekt in eine Fabrik auszulagern.

II.1.3 Model-View-Controller-Architektur

II.1.3.1 Grundlagen

Im Gegensatz zum Domain-Driven Design, bei welchem es sich um ein Muster zum Entwurf von Software handelt, ist das *Model-View-Controller*-Muster (kurz MVC) einen Schritt näher an der tatsächlichen Umsetzung angesiedelt. Dabei handelt es sich um ein sogenanntes *Architekturmuster*, welches eine Anwendung in drei Komponenten aufzuteilen versucht: Das Datenmodell (engl. *Model*), die Darstellung bzw. Präsentation der Daten (engl. *View*) und der Steuerung des Programmablaufs (engl. *Controller*).

Wie wir uns erinnern, sah bereits das *Domain-Driven Design* eine Unterteilung der Anwendung in mehrere Schichten, namentlich Präsentation, Anwendung, Domäne und Infrastruktur vor. Im folgenden Abschnitt wird deutlich werden, dass sich einige dieser Ideen auch im MVC-Muster wiederfinden lassen.

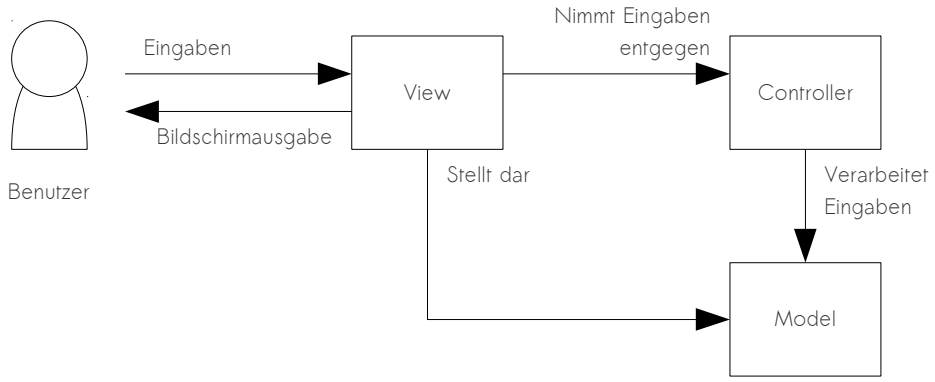

Abbildung 5: Das Model-View-Controller-Muster

II.1.3.2 Datenmodell

Das Datenmodell (engl. *Model*) ist eine Abbildung der von der Anwendung darzustellenden und zu verarbeitenden Daten. Es ist vollkommen unabhängig von einem Controller oder einem View. Das Datenmodell kann je nach Implementierung auch Teile der Geschäftslogik enthalten. In der Regel besteht das Modell aus Klassen verschiedener Typen, und folgt damit einem streng objektorientierten Ansatz.

Um diese Daten dauerhaft (persistent) speichern zu können, werden die Klassen des Datenmodells häufig auf Tabellen in einer Datenbank abgebildet.

Für den Entwurf dieses Datenmodells kann nun das Domain-Driven Design verwendet werden. Wenn wir uns an das Schichtenmodell erinnern, welches vom Domain-Driven Design vorausgesetzt wird, so entspricht das Datenmodell des MVC-Musters genau der Anwendungsdomäne dieser Schichtenarchitektur.

II.1.3.3 Präsentation

Die Präsentationsschicht dient dazu, die Daten aus dem Datenmodell für den Benutzer zu präsentieren, und gegebenenfalls Eingaben von dem Benutzer entgegen zu nehmen. Der View selbst verarbeitet diese Benutzereingaben jedoch nicht, sondern reicht sie weiter an den Controller. Die Präsentation entspricht der Präsentationsschicht, die bereits vom *Domain-Driven Design*-Ansatz vorgeschlagen wurde.

II.1.3.4 Steuerung

Der Controller übernimmt die Steuerung des Programmablaufs. Er reicht die notwendigen Daten an die Präsentationsschicht weiter, und nimmt Benutzereingaben vom View entgegen. Der Controller bearbeitet die Benutzereingaben und reicht sie gegebenenfalls an das Datenmodell weiter.

In der Regel bietet es sich an, einen Controller weiter in sogenannte Aktionen zu unterteilen. Diese Aktionen bilden einzelne Teilbereiche des Controllers ab. FLOW3 und Extbase tragen diesem Umstand Rechnung, indem ein spezieller *ActionController* angeboten wird.

Auch bei komplexen Anwendungen ist es auffällig, dass viele der Aktionen, die ein Benutzer durchführen kann, mit einem einzelnen bestimmten Objekt oder Aggregat der Domäne zu tun haben. Aus diesem Grund macht es zunächst Sinn, jedem Objekt aus dem Domänenmodell, mit welchem ein Benutzer interagieren können soll, einen eigenen Controller zuzuordnen.

Haben wir also ein Domänenobjekt *Kunde*, so brauchen wir auch einen Kunden-Controller. Dieser stellt nun Benutzerschnittstellen für sämtliche Aktionen (engl. *actions*) bereit, die der Benutzer mit diesem Kunden vornehmen kann; beispielsweise Kunden nach einem bestimmten Kriterium zu suchen, alle Details zu einem bestimmten Kunden anzuzeigen, einen neuen Kunden anzulegen, ihn zu kündigen, seine Liefer- oder Rechnungsadresse zu bearbeiten und vieles mehr. Jeder dieser Controller-Aktionen (daher übrigens auch die Bezeichnung *ActionController*) wird dann eine ganz bestimmte View zugeordnet.

So könnte unser Kunden-Controller also eine Aktion *Suchen* enthalten. Dieser Aktion ist ein View zugeordnet, welcher ein Suchformular enthält. Des weiteren enthält der Controller eine Aktion *Neu*, die ein Formular für einen neuen Kunden darstellt und diesen gegebenenfalls speichert.

II.2 Entwurf der Anwendung

Nachdem wir nun unser theoretisches Vorwissen auf den aktuellen Stand gebracht haben, wollen wir dieses nun endlich in der Praxis anwenden. Wie bereits in der Einleitung angerissen, möchten wir ein Zeiterfassungsmodul entwickeln, mit dem Entwickler oder Designer Arbeitszeiten an bestimmten Projekten erfassen können sollen. Zu diesem Zweck werden wir im Lauf des nächsten Abschnittes noch einmal die genauen Anforderungen klären. Anschließend werden wir versuchen, die Prinzipien des Domain-Driven Designs anzuwenden und eine sinnvolle MVC-Struktur auf die Beine zu stellen. Da unsere Daten schließlich in einer Datenbank gespeichert werden sollen, kommen wir auch nicht herum, ein paar Datenbanktabellen für unsere Erweiterung zu entwerfen.

II.2.1 Anforderungen

Entwickelt werden soll ein Zeiterfassungsmodul für Entwickler oder Designer, die projektbezogen arbeiten. In dem Modul sollen beliebig viele Projekte verwaltet werden können, die beliebig untereinander verschachtelt werden können. Jedem Projekt soll sich eine Menge an Mitarbeitern in verschiedenen Rollen (Mitarbeiter und Projektleiter) zuordnen lassen. Anschließend soll jeder Mitarbeiter in einem Projekt, dem er zugewiesen ist, Zeitbuchungen eintragen können. Eine Zeitbuchung wird durch ein Zeitpaar mit Anfangs- und Endzeit sowie einen kurzen Kommentar ausgezeichnet.

II.2.2 Die Anwendungsdomäne

Die Anwendungsdomäne unserer Zeiterfassung muss mehrere verschiedene Objekte abbilden können. Offensichtlich sind zunächst die Mitarbeiter und die Projekte. Außerdem muss es verschiedene Rollen geben, des weiteren müssen die Zeitpaare abgebildet werden können. Für den Entwurf der Domäne bietet sich zunächst ein UML-ähnliches Schema an, da es sich gut zur Visualisierung der Assoziationen zwischen den verschiedenen Objekten eignet.

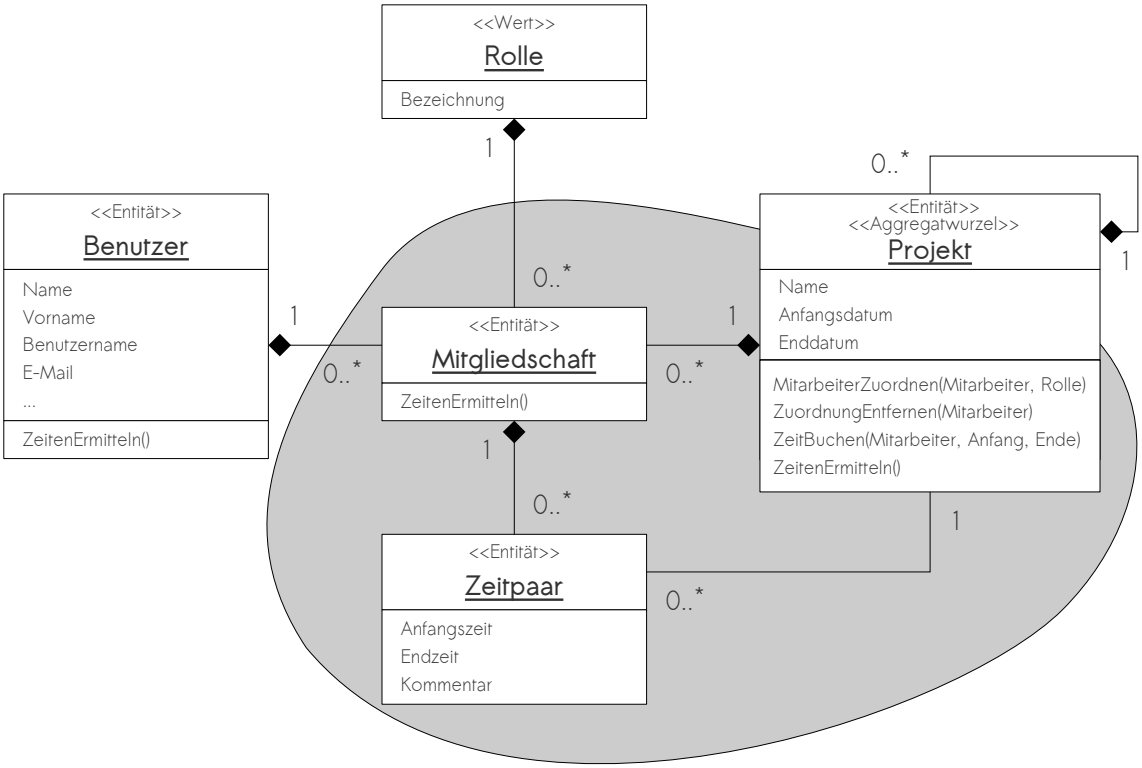

Abbildung 6: Domänenmodell der Zeiterfassungserweiterung

Zunächst macht es Sinn, die Projekte, Mitarbeiter und Zeitpaare als Entitäten zu betrachten, da sie beide sich nicht eindeutig über ihre Attribute identifizieren lassen. Die Rollen sind Wertobjekte, da sie sich ausschließlich über ihre Attribute definieren lassen.

Ein *Projekt* zeichnet sich zunächst durch seine Bezeichnung sowie ein Anfangs- und ein Enddatum aus. Zu den Methoden, die ein Projekt bereitstellen muss, gehören das Zuordnen von Mitarbeitern in bestimmten Rollen, sowie das Entfernen dieser Zuordnungen. Außerdem kann jedes Projekt beliebig viele Unterprojekte haben. Die Projektklasse ist also mit sich selbst assoziiert (eine solche Assoziation wird auch *reflexive Assoziation* genannt).

Ein *Benutzer* zeichnet sich zunächst einmal durch seinen Namen aus. Außerdem kann er einen zusätzlichen Benutzernamen zur eindeutigen Identifizierung sowie diverse Kontaktinformationen haben. Die Benutzer sind in einer *n:m*-Beziehung mit den Projekten assoziiert; dies bedeutet, dass jeder Benutzer beliebig vielen Projekten (oder auch keinem einzigen) zugeordnet sein kann.

Um die Beziehung zwischen Projekten und Benutzern in verschiedenen Rollen abbilden zu können, erweitern wir die Domäne außerdem noch um Mitgliedschaften. Dieses Objekt können wir später auch verwenden, um zum Beispiel die Arbeitszeiten eines einzelnen Mitarbeiters an einem bestimmten Projekt zu ermitteln.

Ein einzelnes *Zeitpaar* wird durch die Attribute Anfangszeit, Endzeit und einen Kommentar ausgezeichnet. Des weiteren ist jedes Zeitpaar genau einer Mitgliedschaft und somit auch einem Projekt und einem Benutzer zugeordnet.

Des weiteren können wir die Projekte, Mitgliedschaften und Zeitpaare als Aggregat betrachten, da jedes Zeitpaar fest einem einzelnen Projekt zugeordnet werden kann. Das Projekt ist dabei die Wurzel des Aggregates. Auf die Zeitpaare kann somit nur per Traversion über das übergeordnete Projekt zugegriffen werden.

Das Domänenmodell ist zu diesem Zeitpunkt noch vollkommen unabhängig von einer konkreten Implementierung oder Programmiersprache. Die Entscheidung, jedes Objekt der Domäne mit einer Klasse und einer Datenbanktabelle abzubilden, drängt sich zwar auf; theoretisch sind jedoch auch andere Implementierungen denkbar (FLOW3 zum Beispiel speichert Objekte statt in einer relationalen Datenbank in einem *Content Repository* nach der JCR-Spezifikation[5]).

Des weiteren war die Verwendung deutschsprachiger Bezeichner in diesem Fall zwar eine bewusste Entscheidung; spätestens bei der konkreten Implementierung sollte man jedoch – allein schon, um unschöne Sprachvermischungen wie *getZeitpaare* zu vermeiden – ausschließlich englischsprachige Bezeichner verwenden. Will man sichergehen, dass keine Übersetzungsschwierigkeiten auftreten, empfiehlt es sich, auch das Domänenmodell bereits in englischer Sprache zu entwerfen.

5 JCR steht für Java Content Repository. Dabei handelt es sich um einen Standard, der ursprünglich aus der Java-Welt stammt. Ein solchen Content Repository ist eine objektorientierte Datenbank, in welcher komplexe Objekte in einer hierarchischen Baumstruktur gespeichert werden können. Da wir dieses Thema an dieser Stelle nicht weiter vertiefen möchten, sei hier auf einige weiterführende Artikel verwiesen. Eine Aufstellung findet sich unter http://wiki.apache.org/jackrabbit/JcrLinks

II.2.3 Klassen und Objekte

II.2.3.1 Datenmodell

Nachdem wir ein Modell der Domäne entwickelt haben, können wir dieses nun in ein Klassendiagramm übersetzen. Dabei entspricht im weitesten ein Domänenobjekt einer Klasse. Eine mögliche Umsetzung stellt das untenstehende Klassendiagramm dar.

Zur Übersichtlichkeit wurden im Diagramm die meisten sondierenden und manipulierenden Methoden nicht eingezeichnet. Da beispielsweise die Attribute der Klasse *User* als *protected* gekennzeichnet sind (in UML-Notation durch ein #-Zeichen dargestellt), muss es logischerweise eine Methode *getUsername()*, *getName()*, *setUsername(name)* usw. geben.

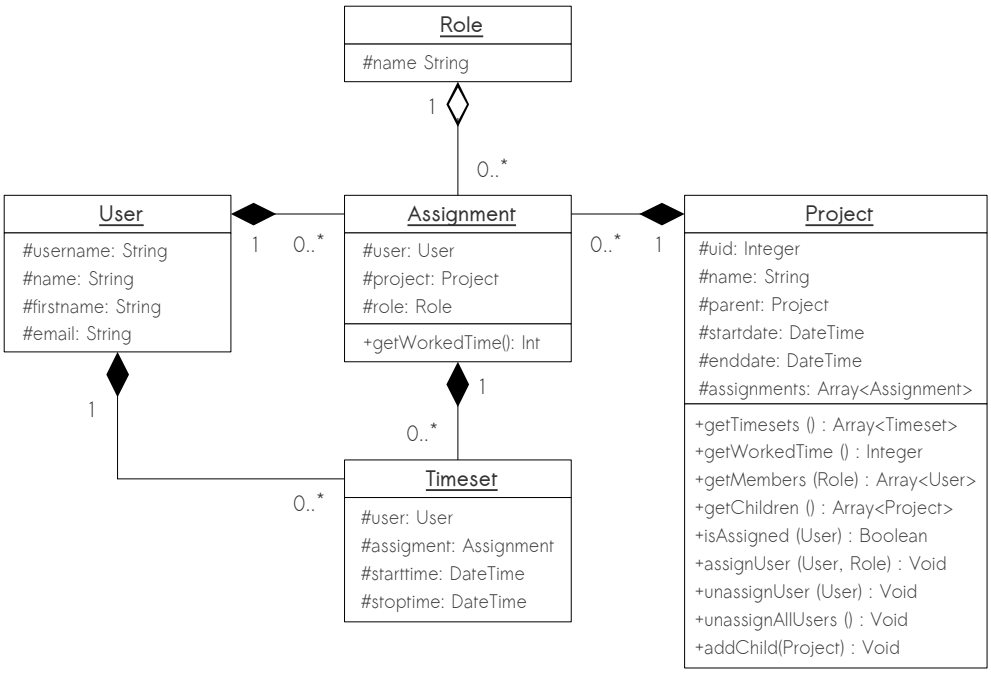

Abbildung 7: Klassendiagramm der Domänenmodells für die Zeiterfassungserweiterung

Außerdem ist die tatsächliche Benennung der Klassen noch nicht endgültig. Bei der Umsetzung unseres Projektes werden wir uns an die Gegebenheiten und Konventionen von Extbase und FLOW3 anpassen müssen.

II.2.3.2 Repositories

Nach unserem Domänenmodell muss ein globaler Zugriff auf alle Projekte, sowie auf alle Benutzer und alle Rollen möglich sein. Auf die Zeitpaare und Mitgliedschaften kann per Traversion zugegriffen werden. Aus diesem Grund muss im nächsten Schritt jeweils ein Repository für die Projekte, die Benutzer sowie die Rollen implementiert werden. Hinsichtlich der Benennung bieten sich *ProjectRepository*, *RoleRepository* und *UserRepository* an. Diese Repositories müssen Methoden implementieren, um etwa alle Projekte/Rollen/Benutzer aus der Datenbank laden zu können, oder z.B. bestimmte Projekte, die einem anderen Projekt untergeordnet sind, oder die einen bestimmten Kennzeichner haben. Glücklicherweise wird uns Extbase bei dieser Implementierung später sehr viel Arbeit abnehmen, weshalb wir das Design der Repositories an dieser Stelle gar nicht weiter vertiefen möchten.

II.2.3.3 Controller

Fehlen nun noch einige Controller, mit denen wir den Programmablauf steuern können. Hier werden wir ausschließlich *ActionController* verwenden, da diese auch von Extbase am besten unterstützt werden. So benötigen wir zunächst einmal einen Controller, um Benutzereingaben an das Projektmodell weiter zu leiten. Welcher Name bietet sich besser an als *ProjectController*?

Dieser Controller sollte eine Aktion zum Darstellen aller verfügbaren Projekte implementieren (*list*), eine Aktion zur Darstellung aller Details über ein bestimmtes Projekt (*show*) sowie Aktionen zum Anlegen, Bearbeiten und Löschen von Projekten (*new*, *edit*, und *delete*). Des weiteren hat es sich eingebürgert, das Darstellen von Formularen und das tatsächlichen Speichern von Daten in unterschiedlichen Aktionen zu organisieren. Fügen wir also außerdem noch eine Aktion *create* zum Speichern von neuen Projekten und eine Aktion *update* zum Speichern von bearbeiteten Objekten hinzu.

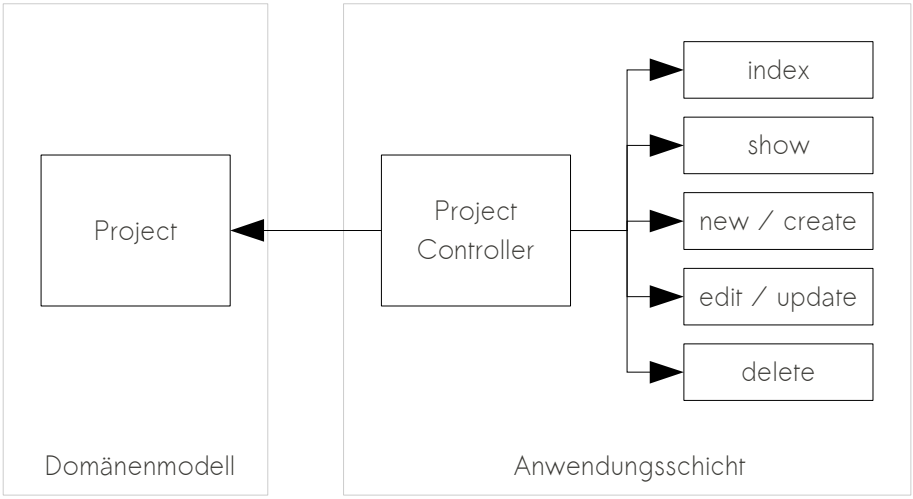

Abbildung 8: Der ProjectController

Zuletzt brauchen wir noch einen *TimesetController*, welcher Aktionen für das Anlegen eines neuen Zeitpaares zur Verfügung stellt. Diese sollen die Namen *new* und *create* tragen. Weitere Aktionen benötigt dieser Zeitpaar-Controller zunächst nicht.

II.2.4 Der Datenbankentwurf

FLOW3 bietet mit dem *Persistence Framework* bereits von Haus aus eine Möglichkeit, Domänenobjekte ohne weitere Konfiguration persistent zu speichern. Bei Extbase hingegen müssen nach wie vor die Datenbanktabellen zusammen mit der Extension angelegt werden. Dazu weist Extbase jeder Klasse des Domänenmodells eine einzelne Datenbanktabelle zu. Referenzen zwischen Objekten werden in der Datenbank als Fremdschlüsselbeziehungen abgebildet. Dieses Vorgehen nennt sich *objektrelationales Mapping*.

Bei Verwendung des Extbase-Kickstarters muss dem Datenbankentwurf nicht viel Beachtung beigemessen werden, da der Kickstarter diesen automatisch aus dem Domänenmodell ableitet. Der Vollständigkeit halber soll an dieser Stelle jedoch dennoch kurz darauf eingegangen werden.

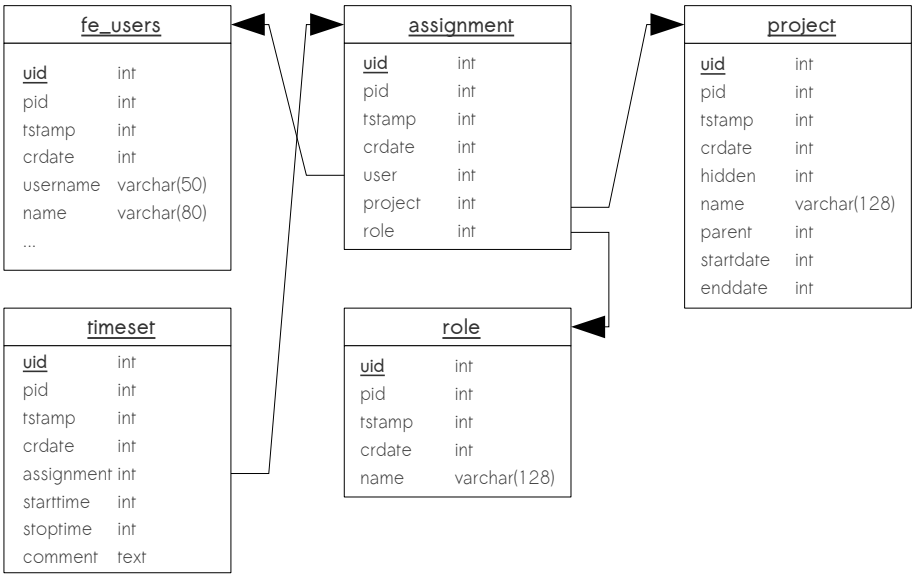

Abbildung 9: Datenbankschema für die Zeiterfassungserweiterung

Genau wie beim Klassendiagramm sind auch hier die Tabellennamen vorläufig. Die Felder *uid*, *pid*, *tstamp*, *crdate* und *hidden* werden von TYPO3 vorausgesetzt. Zur Abbildung der Klasse User wurde hier die bereits im TYPO3-Core existierende *fe_users*-Tabelle verwendet.

Teil III: Erste Schritte in Extbase

Nachdem wir nun lange die theoretischen Grundlagen für die Entwicklung von Erweiterungen mit Extbase besprochen haben, wird es Zeit, die gelernten Konzepte auch in die Praxis umzusetzen. Und nun wird es auch endlich einmal konkret: In diesem Kapitel beschäftigen wir uns nun endlich mit unserer ersten eigenen Erweiterung unter Extbase unter der Verwendung der Template-Engine Fluid.

Dazu möchte ich zunächst noch ein paar allgemeine Worte zum Thema Extbase und Fluid verlieren, und danach kann es auch gleich schon losgehen. Ziel dieses Kapitels ist es, eine lauffähige Zeiterfassungserweiterung mit einem Datenmodell, ein paar Controllern und dazugehörigen Views auf die Beine zu stellen.

III.1 Grundlagen von Extbase

III.1.1 Was ist Extbase?

III.1.1.1 Grundlagen

Bereits vor einiger Zeit wurde von dem Entwicklungsteam für TYPO3 Version 5 um Robert Lemke beschlossen, die Codebasis für TYPO3 5 komplett neu zu entwickeln. Diese Version wird auf einem eigens für diesen Zweck entwickelten Framework, FLOW3, aufbauen. Dabei handelt es sich um ein *Enterprise Application Framework*, welches die Entwicklung großer, anwendungsdomänen-orientierter Applikationen vereinfachen soll.

> **Was ist ein Framework?** Ein Framework ist ein Programmiergerüst, welches bestimmte Programmfunktionen zur Verfügung stellt. Im Gegensatz zu einer herkömmlichen Funktionsbibliothek zeichnet sich ein Framework vor allem durch die sogenannte Steuerungsumkehr (engl. *Inversion of Control*) aus. Dies bedeutet, dass der Programmablauf nicht von der Anwendung selbst gesteuert wird, sondern von dem darunter arbeitenden Framework.

Während FLOW3 nun eine ganze Menge an Entwurfsmustern unterstützt (um mit MVC, *Aspektorientierter Programmierung* und *Test-Driven Development* nur einige zu nennen), bemühen sich Extbase und Fluid, einige dieser Komponenten bereits unter TYPO3 4 zur Verfügung zu stellen. So stellt Extbase das MVC-Gerüst aus FLOW3 zur Verfügung, während Fluid eine mächtige Templating-Engine anbietet. Dazu wurde zum Teil Code aus FLOW3 zurück portiert, und zum anderen Teil per Reverse Engineering nachgebaut, um dieselben Schnittstellen abzubilden. Dies hat vor allem zwei Gründe:

1. Da sich die Schnittstellen sehr ähneln, unterscheidet sich die Entwicklung von Erweiterungen mit Extbase und Fluid nur wenig von der Entwicklung unter FLOW3. Entwickler können sich also bereits jetzt in die neuen Entwurfsmuster einarbeiten.
2. Auf Extbase basierende Erweiterungen sollen beim Erscheinen von TYPO3 5 sehr einfach nach FLOW3 portierbar sein. Damit lässt sich bereits jetzt unter TYPO3 4.3 zukunftssicher entwickeln.

Extbase löst damit die *pi_base*-Klasse ab, welche bisher als Grundlage für eigene Erweiterungen diente, und stellt ein mächtiges MVC-Framework für domänenorientierte Erweiterungen zur Verfügung. Des weiteren entfallen dank der neuen Templating-Engine Fluid große Code-Passagen, die lediglich zum Hin- und Herkopieren von Subparts oder dem Ersetzen von Markern in einer HTML-Vorlage dienen.

III.1.1.2 Installation von Extbase und Fluid

Extbase und Fluid werden seit TYPO3 4.3.0 zusammen mit dem TYPO3-Core als Systemerweiterungen ausgeliefert, sind aber standardmäßig nicht installiert. Beide Erweiterungen können problemlos über den Erweiterungsmanager im TYPO3-Backend nachinstalliert werden:

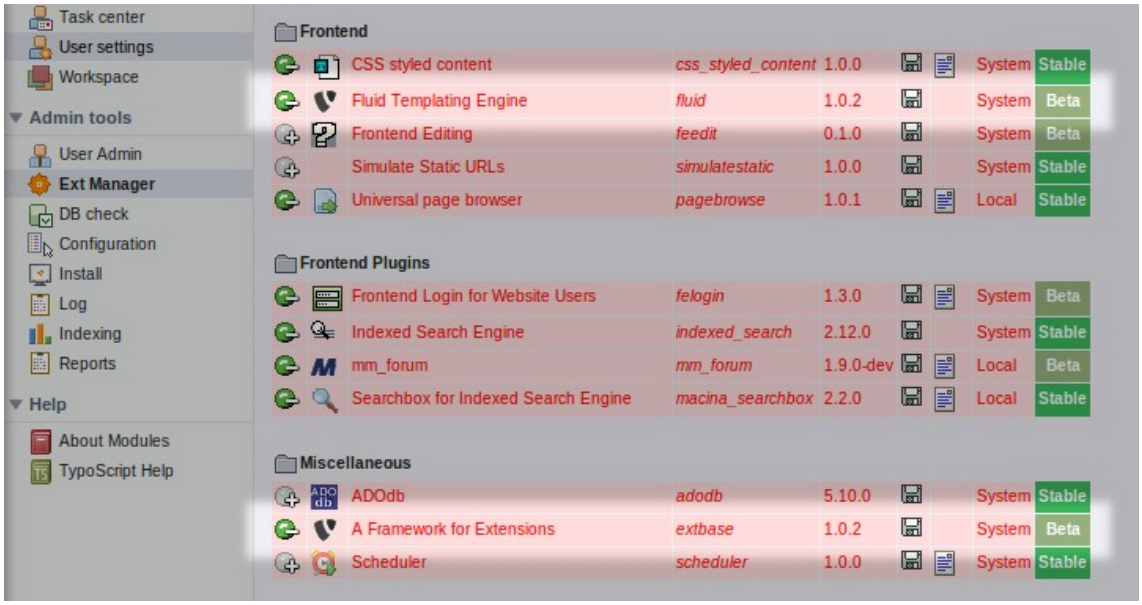

Abbildung 10: Installation von Extbase und Fluid über den Erweiterungsmanager

III.1.2 Convention over Configuration

Einer der Grundsätze, die hinter FLOW3 und Extbase stehen, heißt *Convention over Configuration*, zu deutsch etwa *Konvention steht über Konfiguration*. Dies bedeutet, dass das Framework die meisten Aspekte der Anwendung zunächst einmal „annimmt“, und lediglich davon abweichende Gegebenheiten gesondert konfiguriert werden müssen. Auf diese Weise wird die Menge der zu schreibenden Konfigurationsdateien minimiert.

Eine solche Konvention ist zum Beispiel, dass die einer Klasse zugeordnete Datenbanktabelle denselben Namen trägt. Da das Framework diesen Zusammenhang automatisch annimmt, muss der Entwickler nicht mehr für jede Klasse konfigurieren, wie die dazugehörige Tabelle heißt. Dies ist dann nur noch notwendig, wenn eine Tabelle beispielsweise einen abweichenden Namen trägt.

Eine andere Konvention besagt, dass die Verzeichnisstruktur einer Erweiterung die Namensräume der Klassen widerspiegeln muss, und umgekehrt. Des weiteren muss der Namensraum einer Klasse mit dem Namen der Erweiterung beginnen. Enthält unsere Erweiterung *mittwald_timetrack* also einen Controller für das Domänenobjekt *Project*, so muss diese den Namen *Tx_MittwaldTimetrack_Controller_ProjectController* tragen und unter dem Dateinamen *Controller/ProjectController.php* gespeichert sein.

> **Namensräume:** FLOW3 verwendet zur Benennung von Klassen sogenannte Namensräume (engl. *namespaces*). Da diese erst seit PHP 5.3 zur Verfügung stehen, werden in Extbase diese Namensräume stattdessen durch den Klassennamen abgebildet. Eine Klasse unter FLOW3 könnte also zum Beispiel *\F3\MittwaldTimetrack\Domain\Model\Project* heißen. Hierbei ist *Project* der Klassenname und *\F3\MittwaldTimetrack\Domain\Model* der Namensraum. Unter Extbase würde dieselbe Klasse den Namen *Tx_MittwaldTimetrack_Domain_Model_Project* tragen.

III.1.3 Aufbau einer Extbase-Erweiterung

Werfen wir nun zunächst einmal den Blick auf den Aufbau einer durchschnittlichen Extbase-Erweiterung. Die hier zu sehende Struktur ist von Extbase fest vorgeschrieben:

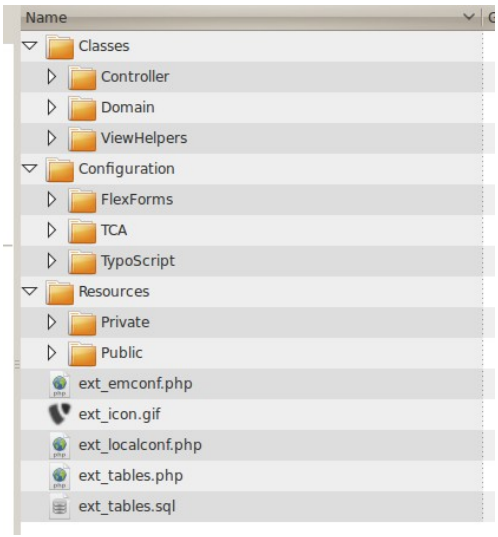

Abbildung 11: Aufbau einer Extbase-Erweiterung

Im Erweiterungsverzeichnis selbst liegen zunächst die grundlegenden Konfigurationsdateien. Da diese nach wie vor vom TYPO3-Kern benötigt werden, hat sich an diesen mit der Einführung von Extbase nicht viel geändert. Der Vollständigkeit halber werfen wir dennoch einen kurzen Blick darauf:

Datei	Beschreibung
ext_emconf.php	Diese Datei beinhaltet Informationen über diese Erweiterung, wie zum Beispiel die Versionsnummer, Angaben zum Autor, oder auch Einstellungen über Abhängigkeiten oder Konflikte gegenüber anderen Erweiterungen. Diese Datei wird vom Erweiterungsmanager ausgelesen.
ext_localconf.php	Die *ext_localconf.php* beinhaltet grundlegende Konfigurationseinstellungen für diese Erweiterung. Sie wird bei jedem Seitenaufruf – sowohl im Frontend als auch im Backend – aufgerufen. Hier wird zum Beispiel die genauere Konfiguration von Frontendplugins vorgenommen.

ext_tables.php	Die *ext_tables.php* enthält weiterführende Konfigurationseinstellungen. Hierzu zählen zum Beispiel die Konfiguration von Tabellen, die zu dieser Erweiterung gehören oder das Registrieren von Frontendplugins oder Backendmodulen.
ext_tables.sql	Diese Datei enthält die Struktur für die Datenbanktabellen, die von dieser Extension benötigt werden.

III.1.3.1 Klassen

Die komplette Programmlogik, einschließlich des Domänenmodells, den Controllern und allen anderen benötigten Klassen, wird im Verzeichnis *Classes/* gespeichert. Die Unterverzeichnisse dieses Ordners sind wie folgt strukturiert:

Verzeichnis	Beschreibung
Controller/	In diesem Verzeichnis sind sämtliche Controller für diese Erweiterung gespeichert. Eine Extbase-Erweiterung muss *mindestens* einen Controller beinhalten, da sonst keine Ausgabe erfolgen kann. Per Konvention endet der Klassennamen jedes Controllers mit *Controller* (also zum Beispiel *ProjectController*).
Domain/	Dieses Verzeichnis beinhaltet die Anwendungsdomäne, welche sich auf weitere Unterverzeichnisse aufteilt. *Domain/Model/* beinhaltet das Domänenmodell. Per Konvention ist jede Klasse genau nach dem Fachobjekt zu benennen, das von ihr abgebildet wird (also zum Beispiel *Project*). Die Klassen in diesem Verzeichnis bilden außerdem das Datenmodell des MVC-Musters. *Domain/Repository/* beinhaltet Repositories für globale Domänenobjekte. Der Name jeder Repository-Klasse muss mit dem Begriff *Repository* enden (Beispiel: *ProjectRepository*). *Domain/Validator/* beinhaltet Service-Objekte, die zur Validierung von Fachobjekten dienen. Mehr über Validatoren findet sich in Abschnitt IV.5.1 auf Seite 143.

Views/	Dieses Verzeichnis enthält Objekte, die die Ausgabe der Erweiterung regeln. Diese Objekte entsprechen der Präsentationsschicht des MVC-Musters. Da Extbase bereits einige vorgefertigte View-Klassen mit sich bringt, ist es meistens nicht einmal nötig, eine eigene View zu implementieren. Dieses Verzeichnis ist somit optional. In Abschnitt IV.2.5 auf Seite 128 werden wir betrachten, wie man eigene View-Klassen schreiben kann.
ViewHelpers/	Das Verzeichnis *ViewHelpers/* beinhaltet Klassen, die zur Unterstützung der Ausgabe dienen, und ist optional. Mehr über ViewHelper findet sich in Abschnitt III.2.3 auf Seite 54 sowie in Abschnitt IV.2.3 auf Seite 121.

III.1.3.2 Konfiguration

Das Verzeichnis *Configuration/* beinhaltet diverse Konfigurationsdateien für die Erweiterung, um die man trotz *Convention over Configuration* leider nicht ganz herum kommt. Diese Dateien teilen sich dabei in drei weitere Unterverzeichnisse auf:

Verzeichnis	Beschreibung
FlexForms/	Dieses Verzeichnis beinhaltet Konfigurationsdateien für FlexForms-Elemente. Diese dienen dazu, zusätzliche Konfigurationsformulare für Plugins darzustellen. Dieses Verzeichnis ist selbstverständlich nur dann erforderlich, wenn auch wirklich ein FlexForms-Formular benötigt wird.
TCA/	Das Verzeichnis TCA/ beinhaltet Dateien zum Einrichten des *Table Configuration Array* (kurz TCA). Dieser beschreibt das Verhalten von Datenbanktabellen.
TypoScript/	In diesem Verzeichnis sind die Typoscript-Konfigurationsdateien für diese Erweiterung gespeichert. In der Regel wird dieses Verzeichnis über einen Eintrag in der *ext_tables.php* als statisches Erweiterungstemplate registriert. Demnach kann dieses Verzeichnis eine Datei *setup.txt* für das Typoscript-Setup sowie eine weitere Datei *constants.txt* für Konstanten enthalten.

III.1.3.3 Ressourcen

Das Verzeichnis *Resources/* beinhaltet sämtliche statischen Ressourcen, die zu dieser Erweiterung gehören. Diese teilen sich in folgende Unterverzeichnisse auf:

Verzeichnis	Beschreibung
Public/	Enthält sämtliche Ressourcen, auf die der Besucher direkt zugreifen können soll. Dies sind zum Beispiel CSS-Dateien oder Bilder. Die weitere Strukturierung dieses Verzeichnisses ist nicht vorgegeben; möglich wären aber zum Beispiel zwei weitere Unterordner *Stylesheets/* und *Images/*.
Private/	Das Verzeichnis *Private/* enthält Ressourcen, auf die der Besucher nicht direkt zugreifen können soll. Diese können also zum Beispiel mit einer .htaccess vor direktem Zugriff geschützt werden. Dieses Verzeichnis ist weiter unterteilt in die Unterverzeichnisse *Templates/*, *Language/* sowie einige weitere optionale.
Private/Templates/	Dieses Verzeichnis beinhaltet die HTML-Vorlagen für die Views dieser Erweiterung. Das *Templates*-Verzeichnis enthält weitere Unterverzeichnisse für jeden Controller der Erweiterung, welche wieder eine Vorlage für jede Aktion des jeweiligen Controllers enthalten.
Private/Language/	Das Language/-Verzeichnis beinhaltet Sprachdateien, die in der Ausgabe verwendet werden können. Diese werden im für TYPO3 üblichen XML-Format gespeichert.
Private/Layout/	Enthält übergeordnete Layouts, die für die Ausgabe verwendet werden sollen. Mehr über Layouts findet sich in Abschnitt IV.2.1 auf Seite 118. Dieses Verzeichnis ist optional.
Private/Partial/	Dieses Verzeichnis enthält Teil-Vorlagen, welche in den Views beliebig wiederverwendet werden können. Auf diese Teilvorlagen wird in Abschnitt IV.2.2 auf Seite 120 näher eingegangen.

III.1.4 Wissenswertes

III.1.4.1 Data-Mapping

Eines der Kern-Features von Extbase ist die Abbildung des Domänenmodells – vorliegend als Sammlung von Klassen – auf eine relationale Datenbank (in der Fachsprache *objektrelationales Mapping* genannt). Die einfachste – und auch am häufigsten angewandte – Methode hierfür besteht darin, jeweils eine Klasse als Tabelle abzubilden, jedes Klassenattribut als eine Spalte und eine Instanz einer Klasse als eine Zeile in der Tabelle. Eben diese Methode wendet auch Extbase an.

Ein Problem besteht jedoch darin, dass die meisten relationalen Datenbankmanagementsysteme – um mit MySQL nur eines zu nennen – in einer Spalte keine sonderlich komplexen Datentypen abbilden können. So wird ein Kalenderdatum in der Regel meistens als Zeichenkette oder Zeitstempel gespeichert; eine Relation zu einer anderen Tabelle über ein Feld, welcher den Primärschlüssel der referenzierten Zeile enthält (ein sogenannter Fremdschlüssel).

Woher weiß Extbase beim Erstellen eines Objektes aus einer Datenbankzeile aber nun, ob es sich bei einer Integer-Spalte um einen Fremdschlüssel, eine Datumsangabe oder schlicht um eine ganz einfache Zahl handelt? Dies festzustellen ist die Aufgabe des *DataMappers*, einer speziellen Komponente von Extbase.

Hierfür wertet der *DataMapper* das sogenannte *Table Configuration Array* für die jeweilige Tabelle aus. An dieser Stelle sollte es genügen, dass die Konfiguration einer Spalte im TCA für die Zuordnung der Werte im späteren Domänenobjekt Ausschlag gebend ist. So werden beispielsweise Spalten, die laut TCA ein Datum oder eine Uhrzeit enthalten sollen, später auch in ein PHP-Objekt vom Typ *DateTime*[6] umgewandelt.

Eine ausführlichere Betrachtung des DataMappers findet sich in Kapitel IV.4.1 aus Seite 139.

6 Bei DateTime handelt es sich um eine PHP-Klasse, die seit Version 5.2 standardmäßig zur Verfügung steht. Siehe hierzu auch die offzielle PHP-Dokumentation unter http://de.php.net/datetime

III.1.4.2 Die Reflection-API

Seit Version 5 ist in PHP eine Erweiterung zur *Reflexion* (engl. *reflection*) enthalten. Diese ermöglicht es, zur Laufzeit Informationen über Klassen, Objekte und Methoden und sogar Quelltextkommentare auszulesen. Dies erlaubt es einer Anwendung, sich selbst über ihre eigene Struktur klar zu werden.

FLOW3 und Extbase nutzen vor allem die Möglichkeit zum Auslesen von Kommentaren, um im eigentlich typenlosen PHP eine Art Typsicherheit herzustellen. Stellen wir uns zur besseren Veranschaulichung eine Beispielmethode vor:

```
Public Function add ($foo, $bar) {
    Return $foo + $bar;
}
```

Da PHP dynamisch typisiert ist, besteht hier zunächst keine Möglichkeit (ohne zusätzliche Typüberprüfungen) um sicherzustellen, dass es sich bei `$foo` und `$bar` tatsächlich um numerische Werte handelt (in einer statisch typisierten Sprache wie Java würde man einfach `int foo, int bar` in den Methodenkopf schreiben). Ergänzen wir diese Methode nun um einen DocComment-Kommentar:

```
    /**
     * Addiert zwei Zahlen
     * @param int $foo Erster Summand
     * @param int $bar Zweiter Summand
     * @return int Summe
     */
Public Function add ($foo, $bar) {
    Return $foo + $bar;
}
```

Wie wir sehen, befindet sich innerhalb des Kommentars eine Angabe bezüglich der Datentypen von `$foo` und `$bar`. Mithilfe der Reflexions-Schnittstelle kann Extbase (und auch FLOW3) dieses Kommentar nun auswerten und so ermitteln, welche Datentypen als Parameter der Methode erwartet werden. Bei Controller-Action-Methoden zum Beispiel hängt sich Extbase vor dieser Methode ein und führt jeweils eine Typüberprüfung durch. Wird nun ein Parameter eines anderen Typs (zum Beispiel ein String) an diese Methode übergeben, so gibt Extbase automatisch eine Fehlermeldung aus.

Und es kommt noch besser: Extbase gibt auch eine Fehlermeldung aus, falls einer der Parameter in dem Kommentar überhaupt nicht erwähnt wird! Insbesondere bei Controller- und ViewHelper-Klassen wird hier sehr rigoros vorgegangen. Diese Art der Dokumentation wird auch für Klassenattribute vorausgesetzt, da ansonsten kein Data-Mapping möglich ist.

Auf diese Weise wird der Entwickler geradezu dazu gezwungen, seinen Programmquelltext zumindest rudimentär zu dokumentieren. Jemandem, der es eilig hat, mag dies zwar wie ein lästiges Ärgernis vorkommen, dennoch sind die Vorteile nicht von der Hand zu weisen:

1. Es wird sichergestellt, dass der Quelltext wenigstens ansatzweise dokumentiert ist. Dies macht eine spätere Wartung um vieles einfacher.
2. Der Entwickler muss sich nicht mehr um immer wiederkehrende Typüberprüfungen kümmern, die unter Umständen vergessen werden und später zu Laufzeitfehlern führen könnten.

III.2 Einführung in Fluid

Bevor wir nun unsere erste Extension anlegen, werfen wir noch einen kurzen Blick auf die Templating-Engine Fluid, und wie sie funktioniert. Um uns noch einmal die Unterschiede zur Entwicklung mit *pi_base* klarzumachen, werden wir daher zunächst noch einmal das bisherige Muster betrachten und uns anschließend voll und ganz Fluid zuwenden.

III.2.1 Ein Blick zurück

Bisherige Erweiterungen für TYPO3 verwenden eine Marker-basierte Schnittstelle für HTML-Vorlagen. Eine derartige Vorlage sieht zum Beispiel so aus:

```
<h2>###LANG_CUSTOMERS###</h2>
<table>
    <tr>
        <th>###LANG_CUSTOMER_NUMBER###</th>
        <th>###LANG_CUSTOMER_NAME###</th>
    </tr>
    <!-- ###CUSTOMER_SUBPART### begin -->
        <tr>
            <td>###CUSTOMER_NUMBER###</td>
            <td>###CUSTOMER_NAME###</td>
        </tr>
    <!-- ###CUSTOMER_SUBPART### end -->
</table>
```

Anschließend ist es notwendig, jedem der verwendeten Marker einzeln einen Inhalt zuzuweisen. Bei der Verwendung verschachtelter Vorlagen verkomplizierte sich die Umsetzung zusätzlich:

```
$template         = $this->cObj->fileResource($templateFile);
$customerTemplate = $this->cObj->getSubpart ( $template,
                                               '###CUSTOMER_SUBPART###' );
$customerContent = '';

ForEach($customers As $customer) {
    $customerMarkers = Array (
        '###CUSTOMER_NUMBER###' => $customer['customernumber'],
        '###CUSTOMER_NAME###'   => htmlspecialchars($customer['name'])
    );
    $customerContent .= $this->cObj->substituteMarkerArray($customerTemplate,
                                                           $customerMarkers);
}
```

```
$markers = Array (
    '###LANG_CUSTOMERS###'       => $this->pi_getLL('customers'),
    '###LANG_CUSTOMER_NUMBER###' => $this->pi_getLL('customer.number'),
    '###LANG_CUSTOMER_NAME###'   => $this->pi_getLL('customer.name')
);
$template = $this->cObj->substituteMarkerArray($template, $markers);
$template = $this->cObj->substituteSubpart ( $template,
                                             '###CUSTOMER_SUBPART###',
                                             $customerContent );
Return $template;
```

Ein unangenehmer Effekt dieser Methode ist, dass eine nicht unerhebliche Menge Code lediglich dafür aufgewendet werden muss, nur um die Marker in der HTML-Vorlage korrekt zu befüllen. Des weiteren findet keine saubere Trennung zwischen Präsentations- und Anwendungslogik statt.

Durch die Verwendung von Fluid entfällt nun ein Großteil dieses Aufwandes. Vor allem das aufwändige Füllen der Marker wird in Zukunft nicht mehr notwendig sein.

III.2.2 Objektorientierte Vorlagen

Wie wir wissen, arbeitet Extbase streng objektorientiert. Jedem Fachobjekt aus der Domäne ist also eine PHP-Klasse zugeordnet. So ist es also nur konsequent, dass auch Fluid diesen objektorientierten Ansatz unterstützt. In diesem Sinne ist es nun möglich, einfach ein komplettes Objekt an die Vorlage zu übergeben. Innerhalb der Vorlage kann dann auf alle sondierenden Methoden dieses Objektes zugegriffen werden. Des weiteren unterstützt Fluid einfache Kontrollstrukturen, wie Schleifen und Bedingungen.

Nachfolgend also noch einmal das Beispiel aus dem letzten Abschnitt, nun mit Fluid:

```
<h2>{f:translate( key : 'customers' )}</h2>
<table>
    <tr>
        <th>{f:translate( key: 'customer.number' )}</th>
        <th>{f:translate( key: 'customer.name' )}</th>
    </tr>
    <f:for each="{customers}" as="customer">
        <tr>
            <td>{customer.number}</td>
            <td>{customer.name}</td>
        </tr>
    </f:for>
</table>
```

Der dazugehörige PHP-Code im Controller lautet folgendermaßen:

```
$view->assign('customer', $customer);
```

Wie wir also sehen, arbeitet Fluid nicht mehr mit den altbekannten Markern. Stattdessen werden nun Anweisungen, die in geschweiften Klammern stehen, dynamisch ersetzt. Wird dem View über die *assign($key, $value)*-Methode ein Objekt zugewiesen, kann per Traversion auf sämtliche Attribute dieses Objektes zugegriffen werden. Zu beachten ist lediglich, dass das Objekt, welches dem View zugewiesen wird, für jedes auszulesende Attribut eine sondierende Methode anbietet. Steht in dem View also die Anweisung `{customer.number}`, so übersetzt Fluid intern diese Anweisung in `$customer->getNumber()`. Diese Anweisungen lassen sich beliebig tief verschachteln. So wird die Anweisung `{customer.billingAddress.zip}` zum Beispiel in `$customer->getBillingAddress()->getZip()` übersetzt.

Des weiteren ist auffällig, dass Fluid auch komplexe Anweisungen versteht, wie zum Beispiel die `f:translate`-Anweisung, um dynamische Sprachinhalte auszulesen. Außerdem ist es möglich, Kontrollstrukturen wie eine For-Each-Schleife zu verwenden.

III.2.3 ViewHelper

Eines der mächtigsten Werkzeuge von Fluid sind die sogenannten ViewHelper. Dies sind zusätzliche Komponenten, die innerhalb eines Views über XML-ähnliche Tags angesprochen werden konnten. Ein solcher ViewHelper ist zum Beispiel der *For*-ViewHelper, den wir bereits im vorherigen Abschnitt verwendet haben. Für komplexere Darstellungen ist es sogar möglich, eigene ViewHelper zu implementieren, deren Verhalten vollkommen frei gesteuert werden kann (siehe hierzu Kapitel IV.2.3 auf Seite 121).

Die nachfolgende Liste stellt einen kurzen Überblick über die gebräuchlichsten ViewHelper dar. Bewusst außen vor gelassen wurden dabei die ViewHelper zur Erzeugung von Formularen. Diesen ist mit Abschnitt III.4.5 auf Seite 90 ein eigenes Kapitel gewidmet.

Eine vollständige Referenz aller ViewHelper findet sich im Anhang V.1 auf Seite 156.

ViewHelper	Beschreibung	Beispiel

If/Then/Else	Überprüft, ob eine Bedingung erfüllt ist. Durch die Verwendung des Then- und Else-ViewHelpers lässt sich auch eine noch differenziertere Unterscheidung treffen.	`<f:if condition="3>1">`
		`  3 ist größer als 1`
		`</f:if>`
		
		`<f:if condition="1==2">`
		`  <f:then>`
		`  1 ist gleich 2`
		`  </f:then>`
		`  <f:else>`
		`  1 ist nicht gleich 2`
		`  </f:else>`
		`</f:if>`

For	Ermöglicht es, eine Liste von Objekten in einer Schleife zu durchlaufen.	`<f:for each="{customers}"` `  as="customer" >` `  {customer.name} ` `</f:for>`
Translate	Liefert einen sprachabhängigen Text zurück. Diese werden aus den XML-Dateien in *Resources/Private/Language* geladen.	`<f:translate key="myKey" />`
Image	Gibt ein Bild aus und skaliert es gegebenenfalls.	`<f:image src="test.png"` `  width="50" />`

Link.Action	Erzeugt einen Link zu einer bestimmten Controller-/Action-Kombination.	`<f:link.action action="index"`
		`  controller="Customer"`
		`  arguments="{page:1}" >`
		`  Link`
		`</f:link.action>`

Link.Page	Erzeugt einen Link zu einer anderen Seite innerhalb des TYPO3-Seitenbaums	`<f:link.page pageUid="123"`
		`additionalParams="{foo:bar}">`
		`  Klick mich!`
		`</f:link.page>`

Format.Nl2br	Wandelt einfache Zeilenumbrüche (Zeilenvorschübe) in HTML-Zeilenumbrüche (` `) um.	`<f:format.nl2br>` `  Hallo Welt` `  Nächste Zeile` `</f:format.nl2br>`

Format.Date Formatiert eine Datumsangabe.

```
<f:format.date
    format="d.m.Y H:i">
  1265798455
</f:format.date>
```

III.3 Anlegen der Extension

Beginnen wir nun damit, unsere Zeiterfassungserweiterung auf Basis von Extbase umzusetzen. Als Extension-Key haben wir dafür an dieser Stelle *mittwald_timetrack* gewählt. Für die ersten Schritte, können wir den Extbase-Kickstarter verwenden. Dieser dient dazu, dem Entwickler die Standardaufgaben beim Anlegen einer neuen Extension abzunehmen.

Leider befindet sich der Extbase-Kickstarter derzeit (September 2010) noch in einem sehr frühen Entwicklungsstadium. Wir werden also zwischendurch einige Unzulänglichkeiten des Kickstarters feststellen, von denen wir uns aber nicht aufhalten lassen möchten.

Statt den Kickstarter zu verwenden, können auch sämtliche benötigten Dateien von Hand angelegt werden. Eine Anleitung hierzu findet sich im Vertiefungsabschnitt in Kapitel IV.1 ab Seite 108.

III.3.1 Installation des Extbase-Kickstarters

Da sich der Kickstarter noch in einem recht frühen Stadium der Entwicklung befindet, steht er noch nicht über das TYPO3 Extension Repository (TER) zur Verfügung. Stattdessen haben wir die Möglichkeit, ihn über das entsprechende Sourceforge-Projekt[7] oder gleich über das offizielle Subversion-Repository[8] zu beziehen.

Nach der Installation des Kickstarters über den Erweiterungsmanager nistet sich dieser im Admin-Bereich des TYPO3-Menüs ein.

7 Siehe http://sourceforge.net/projects/typo3xdev/files/T3X/extensions/extbase_kickstarter-trunk_typo3xdev.t3x/download

8 Siehe https://svn.typo3.org/TYPO3v4/Extensions/extbase_kickstarter/trunk

III.3.2 Kickstarting der Erweiterung

III.3.2.1 Einstieg

Der Extbase-Kickstarter schreibt einen dreiteiligen Arbeitsablauf vor; dieser beginnt zunächst mit dem Entwurf der Anwendungsdomäne. Hierfür stellt der Kickstarter eine grafische Oberfläche zur Verfügung, die einem UML-Editor ähnelt. Die Domänenmodelllierung kann mit der entsprechenden Schaltfläche in der Start-Ansicht des Kickstarters gestartet werden.

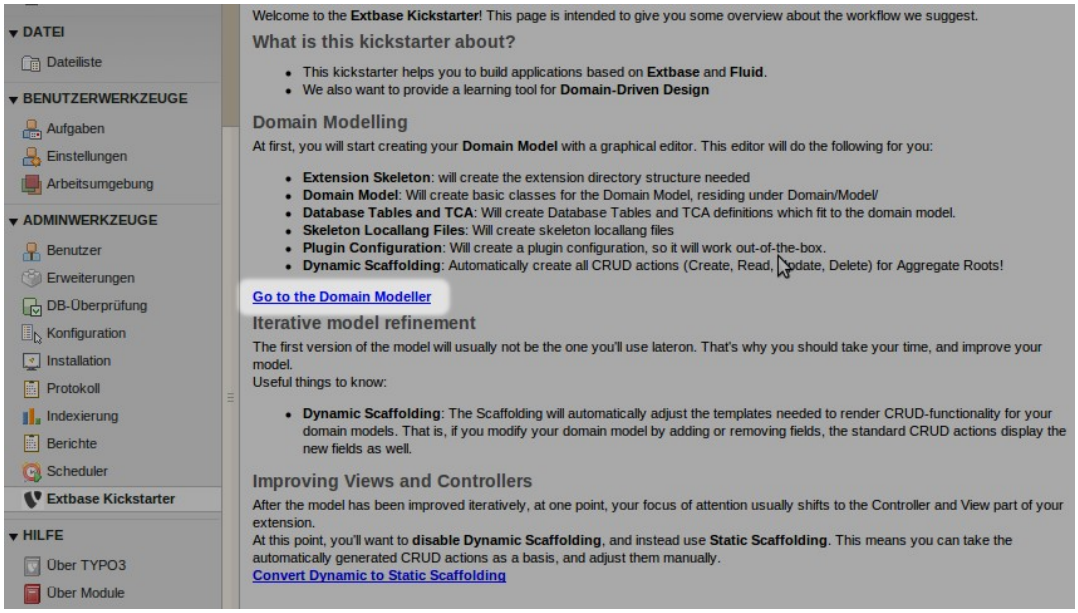

Abbildung 12: Auswahl des Domainmodellierungs-Arbeitsschrittes im Kickstarter

In der nun dargestellten grafischen Oberfläche können die verschiedenen Domänenobjekte per Drag & Drop angeordnet und verknüpft werden. Bevor damit begonnen wird, sollten jedoch zunächst einige grundlegende Angaben zur neuen Extension gemacht werden. Das Formular hierfür versteckt sich als ausklappbarer Bereich am linken Rand des Modellierungswerkzeuges.

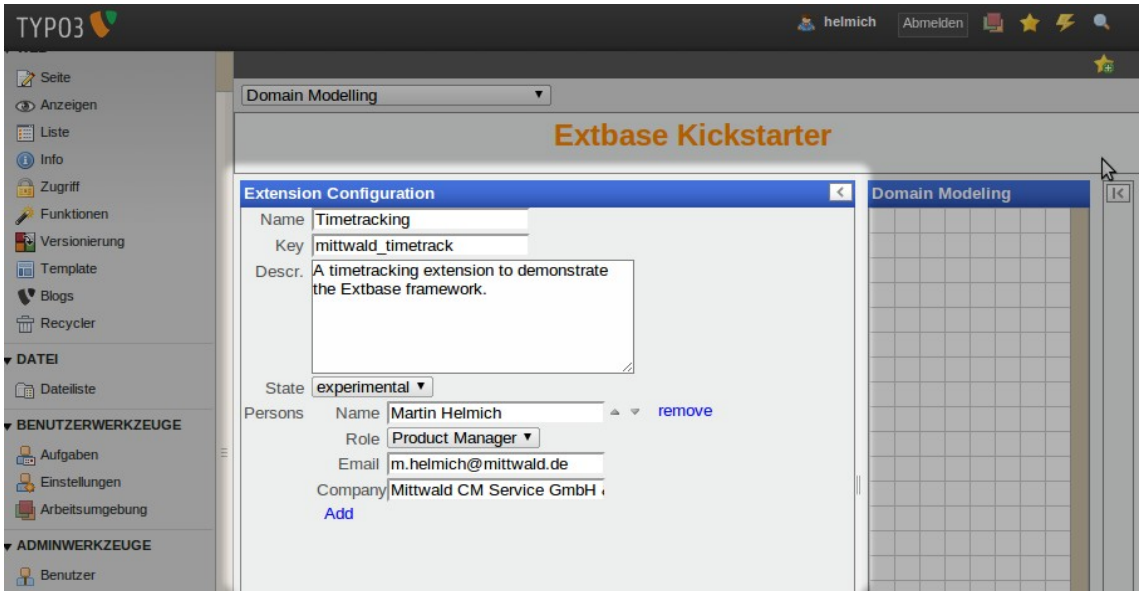

Abbildung 13: Eingabe grundlegender Extension-Daten im Kickstarter

III.3.2.2 Erstellen der Domänenobjekte

Ist diese lästige Pflichtarbeit getan, können die ersten Domänenobjekte im Bearbeitungsbereich platziert werden. Neue Objekte können platziert werden, indem sie von der Schaltfläche „*New Model Object*“ auf die Bearbeitungsfläche gezogen werden.

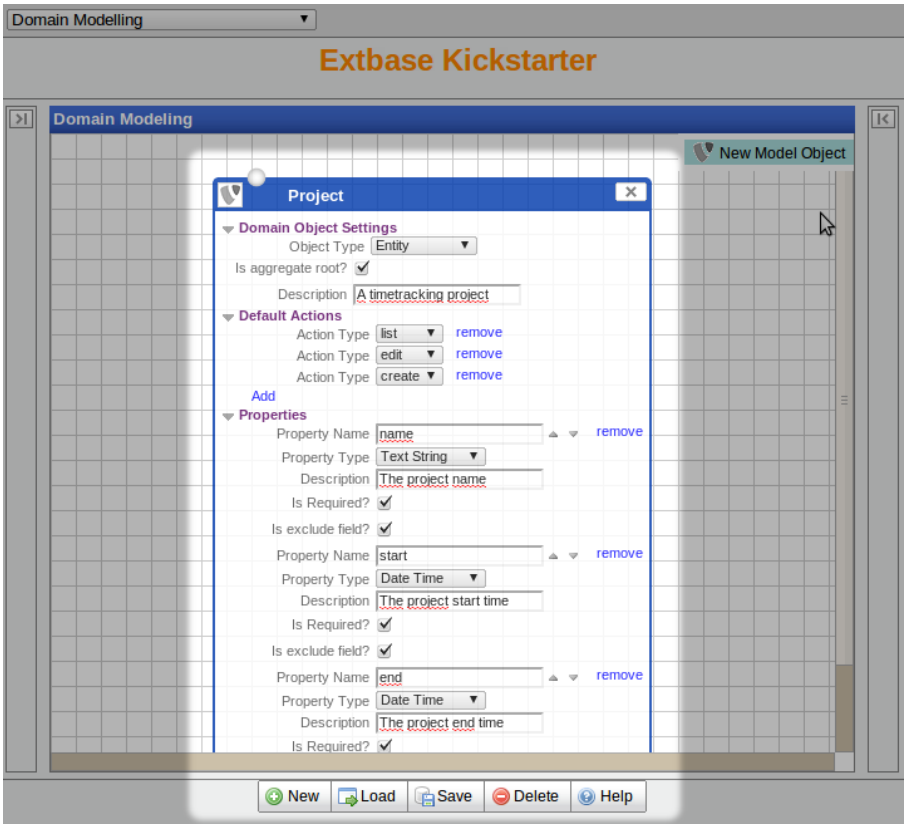

Abbildung 14: Definition eines Domänenobjektes im Kickstarter

Erstellen wir ein neues Domänenobjekt, so sollten wir durch einen Klick auf die Kopfleiste zunächst den Namen für dieses Objekt angeben. Wir beginnen an dieser Stelle mit dem *Project*-Domänenobjekt. Nachdem wir das neue Objekt erstellt und benannt haben, bietet der Kickstarter für dieses Objekt mehrere Einstellungsmöglichkeiten an:

1. Unter den *Domain Object Settings* (Domänenobjekteinstellungen) können allgemeine Einstellungen für dieses Objekt festgelegt werden. Dazu gehört beispielsweise, ob das Objekt eine Entität oder ein Wertobjekt ist, ob es sich um eine Aggregatwurzel handelt sowie eine geeignete Beschreibung. Für das *Project*-Objekt wählen wir nun zunächst die Entität als Objekttyp und legen das Objekt als Aggregatwurzel fest.

2. In dem Abschnitt *Default Actions* (Standardaktionen) können einige Standardaktionen festgelegt, für die der Kickstarter gleich eine Controller-Aktion anlegen soll. Hier werden die Aktionen *list*, *create* und *edit* zur Auswahl angeboten. Für das *Project*-Objekt nutzen wir dies natürlich aus und wählen alle drei Aktionen an.

3. Unter dem Abschnitt *Properties* (Eigenschaften) können wir nun schließlich die Attribute des Objektes angeben. Jedes Attribut definiert sich über einen Namen und einen Typ, der über ein Dropdown-Feld ausgewählt werden kann. Wir legen hier zunächst die Attribute *name*, *start* und *end* des *Project*-Objektes an.
4. Der Abschnitt *Relations* (Relationen) bietet schließlich die Möglichkeit, Beziehungen dieses Objektes zu anderen Domänenobjekten zu definieren. Da das *Project*-Objekt derzeit noch das einzige Objekt ist, überspringen wir diesen Punkt zunächst einmal.

Mit den übrigen Domänenobjekten (*Role*, *Assignment* und *Timeset*) kann analog verfahren werden. Zu beachten ist jedoch, dass bei den Attributen keine Attribute angegeben werden, die Verknüpfungen zu anderen Objekten darstellen; dies erfolgt erst im nächsten Schritt.

III.3.2.3 Verknüpfen der Objekte

Nachdem nun sämtliche Domänenobjekte erstellt wurden, können diese im nächsten Schritt nun miteinander verknüpft werden. Die Relationen zwischen Objekten werden im *Relations*-Abschnitt jedes Objektes angegeben.

Um nun die Relation zwischen dem *Project*- und dem *Assignment*-Objekt herzustellen, fügen wir dem *Project*-Objekt eine neue Relation hinzu und benennen diese mit dem Namen *assignments*. Per Drag & Drop kann nun die Verknüpfung zum *Assignment*-Objekt hergestellt werden. Da es sich um eine 1:n-Beziehung (Ein Projekt zu *n* Mitgliedschaften) handeln soll, wählen wir als Relationstyp *„0..* (foreign Key)“*.

Um nun nicht nur alle Mitgliedschaften eines bestimmten Projektes ermitteln zu können, sondern auch das Projekt einer bestimmten Mitgliedschaft, legen wir nun dieselbe Beziehung noch einmal in umgekehrter Richtung an. Auf diese Weise werden alle Domänenobjekte so wie zuvor festgelegt verknüpft.

> **Hinweis:** Derzeit (September 2010) gibt es im Kickstarter keine Möglichkeit, Relationen zur fe_user-Tabelle abzubilden - wie es in der Zeiterfassungs-Erweiterung beispielsweise bei den Mitgliedschaften notwendig wäre.[9] Diese Beziehungen müssen im Anschluss manuell hinzugefügt werden.

9 Siehe http://forge.typo3.org/issues/9573

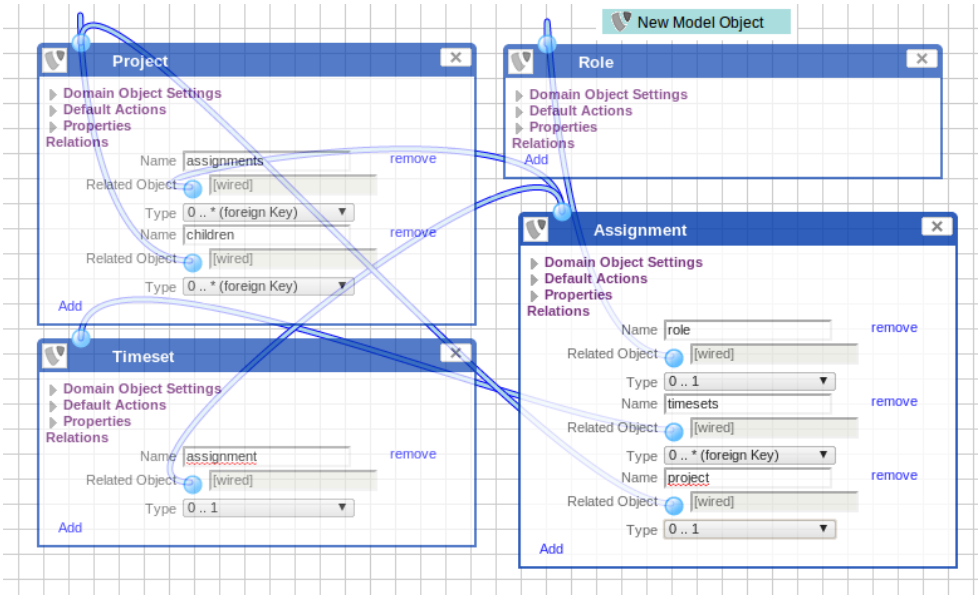

Abbildung 15: Relationen zwischen Objekten im Kickstarter

Mit einem Klick auf *Save* kann der bisherige Stand gespeichert werden. Zu diesem Zeitpunkt werden auch bereits sämtliche Dateien und Ordner im *typo3conf/*-Verzeichnis erstellt.

III.3.2.4 Manuelle Anpassungen

Nachdem der Kickstarter nun das Grundgerüst der Extension erstellt hat, kann nun die Feinarbeit erfolgen. Wie oben bereits erwähnt, kann der Kickstarter keine Relationen zu Objekten anderer Erweiterungen abbilden – in unserem Fall müssen wir jedoch eine Beziehung zwischen dem *Assignments*-Objekt und der *fe_users*-Tabelle abbilden.

Hierzu müssen wir die entsprechende Beziehung – nennen wir sie *user* – im TCA der *assignments*-Tabelle nachtragen. Dieses wird in der Datei *Configuration/TCA/Assignment.php* gespeichert. Dort fügen wir innerhalb des `columns`-Unterarrays folgende Zeilen ein:

PHP: Configuration/TCA/Assignment.php

```
'user' => Array (
    'exclude' => 1,
    'label'   => 'LLL:EXT:mittwald_timetrack/Resources/Private/Language/'
                 . 'locallang_db.xml:'
                 . 'tx_mittwaldtimetrack_domain_model_assignment.user',
    'config'  => Array (
        'type'          => 'select',
```

```
            'foreign_class' => 'Tx_Extbase_Domain_Model_FrontendUser',
            'foreign_table' => 'fe_users',
            'maxitems'      => 1 ) ),
```

Des weiteren wurden im vorigen Schritt z.T. einige bidirektionale Relationen definiert, beispielsweise von *Project* nach *Assignment* und umgekehrt. Der Kickstarter erkennt diese zur Zeit noch nicht als eine einzelne Relation, sondern behandelt diese beide getrennt. Wurde die Relation zum *Project*-Objekt also etwa mit *project* benannt, so erzeugt der Kickstarter zwei *project*-Attribute in der späteren Klasse.

Um diesen Fehler wieder auszugleichen, sollte nun aus der Datei *Configuration/TCA/Assignment.php* folgender Codeabschnitt **entfernt** werden:

PHP: Configuration/TCA/Assignment.php

```
'project' => array(
    'config' => array(
        'type' => 'passthrough'
    )
),
```

Ebenso sollte aus der Datei *ext_tables.sql* aus der Definition der Tabelle *tx_mittwaldtimetrack_domain_model_assignment* folgende doppelt vorhandene Zeile entfernt werden:

SQL: ext_tables.sql

```
project int(11) unsigned DEFAULT '0' NOT NULL,
```

III.3.2.5 Dynamisches und statisches Scaffolding

Würden wir die Erweiterung nun über den Erweiterungsmanager installieren und auf einer Seite als Inhaltselement einfügen, würde der Kickstarter bereits einige vordefinierte Ansichten und Formulare generieren, über welche die Domänenobjekte bearbeitet werden. Diese Ansichten werden dynamisch erstellt und ändern sich, sobald Änderungen an der Struktur des Domänenmodells vorgenommen werden. Dieses Prinzip wird als *Dynamic Scaffolding* bezeichnet (da eine Übersetzung dieses Begriffes eher holperig klingen würde, bleiben wir im weiteren Verlauf bei dem englischsprachigen Begriff).

Unter *Scaffolding* (zu dt. etwa Gerüstbau) versteht man das Erstellen vorläufiger, rudimentärer Views für eine Anwendung, wie beispielsweise eine Listenansicht oder Bearbeitungsformulare.

Der letzte Arbeitsschritt des Kickstarters sieht nun vor, die dynamisch erstellten „Gerüste“ in statische umzuwandeln (*Static Scaffolding*). Bei diesem Arbeitsschritt werden die zuvor dynamisch erstellten Views in statische Template-Dateien konvertiert, die anschließend ganz normal bearbeitet werden können.

Hierfür wählen wir im Kickstarter den letzten Arbeitsschritt *Convert Dynamic to Static Scaffolding*. Im Anschluss präsentiert der Kickstarter eine Liste aller Controller der soeben erstellten Erweiterung. Mit einem Klick auf die entsprechende Schaltfläche können die Aktionen und Views jedes Controllers vom dynamischen zum statischen Scaffolding konvertiert werden.

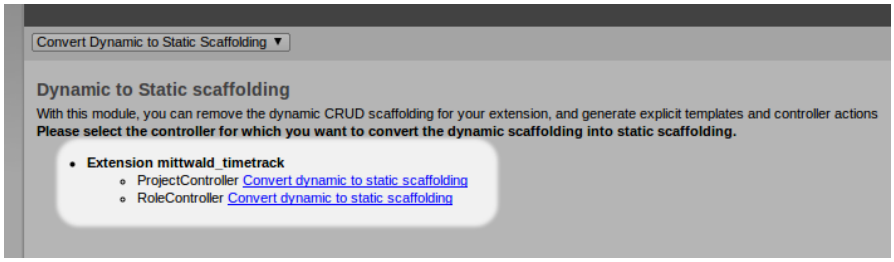

Abbildung 16: Dynamisches und statisches Scaffolding

III.3.3 Installation der Erweiterung

Auch wenn unsere neue Erweiterung noch keine wirkliche Programmlogik enthält, so können wir sie dennoch schon einmal über den Extensionmanager installieren. Hier können wir auch gleich überprüfen, ob alle Einstellungen in der *ext_emconf.php* korrekt sind.

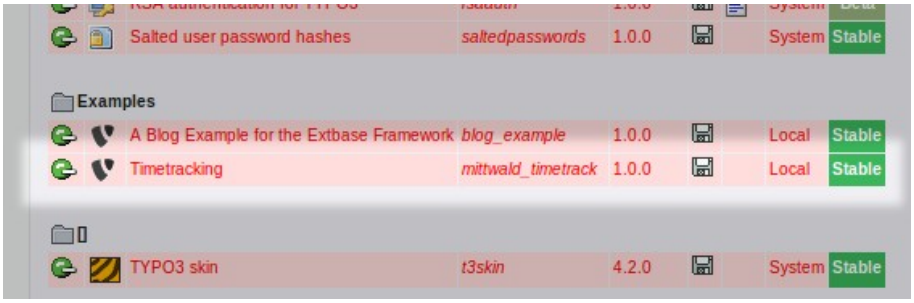

Abbildung 17: Installation der Beispielerweiterung über den Erweiterungsmanager

Bei der Installation sollten nun – soweit noch nicht geschehen – Extbase und Fluid installiert, sowie unsere Datenbanktabellen angelegt werden.

Da wir bereits ein Plugin angemeldet und konfiguriert haben, können wir dieses nun auch sofort als Seiteninhalt auf einer Seite unserer Wahl einfügen:

1. Legen Sie zunächst einen SysOrdner an, in welchem die Zeiterfassungsdaten später gespeichert werden können.
2. Als nächstes legen Sie auf einer Seite unserer Wahl ein neues Inhaltselement vom Typ „Plugin“ an und wählen die Zeiterfassungserweiterung aus. Als „Ausgangspunkt“ des Plugins wählen Sie den soeben erstellten SysOrdner.

> **Hinweis:** Das Setzen des Ausgangspunktes sollte auf jeden Fall beachtet werden; da Extbase diesen automatisch auswertet, um die Objekte aus der Datenbank wieder herzustellen.

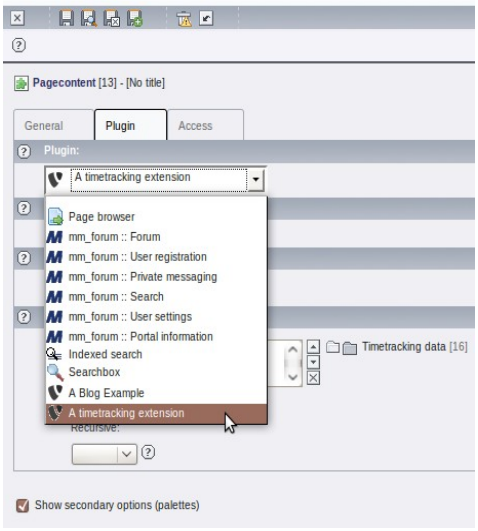

Abbildung 18: Anlegen eines Inhaltselementes mit der neuen Erweiterung

3. Da sämtliche Tabellen im TCA – Kickstarter sei Dank – bereits korrekt konfiguriert sind, können wir über das TYPO3-Backend nun bereits einige Objekte anlegen. Wechseln wir also auf unseren SysOrdner und in das Listen-Modul und legen einige Benutzerrollen, Benutzer und Projekte an.

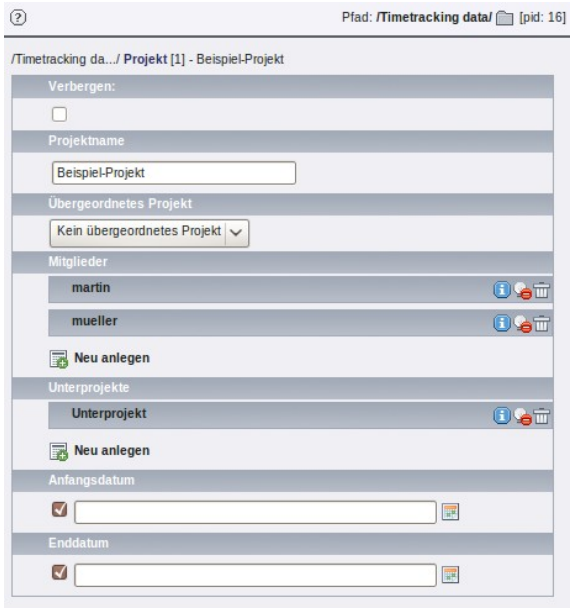

Abbildung 19: Das Bearbeitungsformular für Projekte

III.4 Erste Schritte

Im nächsten Abschnitt beginnen wir nun mit der Implementierung der für unsere Erweiterung erforderlichen Klassen. Dabei beginnen wir zunächst mit der Anwendungsdomäne, bestehend aus dem Datenmodell und den Repositories. Anschließend werden wir einen ersten Controller und einige Views hinzufügen.

Wie bereits angesprochen, müssen die Dateipfade und der Klassenname per Konvention übereinstimmen. Der Dateiname entspricht dabei der letzten Komponente des Klassennamens nach dem letzten „_"-Zeichen. Die Klasse *Tx_MittwaldTimetrack_Domain_Model_Project* würde also in der Datei *Project.php* im Verzeichnis *Classes/Domain/Model/* gespeichert werden.

Die meisten Klassen dürften mit den grundlegenden Attributen und Methoden bereits durch den Kickstarter angelegt worden sein, sodass wir nur noch leichte Veränderungen vornehmen müssen. Um ein besseres Verständnis über die Funktionsweise von Extbase vermitteln zu können, werden wir hier dennoch die Erstellung der benötigten Dateien und Klassen im Detail betrachten.

III.4.1 Implementierung des Domänenmodells

Sämtliche Klassen, die zur Anwendungsdomäne gehören, werden innerhalb des Verzeichnisses *Classes/Domain* gespeichert. Dabei enthält *Classes/Domain/Model* die Klassen für das Datenmodell und *Class/Domain/Repository* die Repositories. Später werden noch weitere Unterverzeichnisse hinzukommen. Doch dazu zu gegebener Zeit mehr.

III.4.1.1 Datenmodell

Extbase stellt bereits einige abstrakte Basisklassen bereit für die Implementierung des Modells bereit. Damit die Werte aus der Datenbank auf unsere Objekte zugewiesen werden können, ist es auch wichtig, diese Klassen tatsächlich zu verwenden. Abhängig davon, ob wir eine Entität oder ein Wertobjekt modellieren wollen (wir erinnern uns an Abschnitt II.1.2.3.a auf Seite 25), stehen hierfür die Klassen *Tx_Extbase_DomainObject_AbstractEntity* sowie *Tx_Extbase_DomainObject_AbstractValueObject* als Basis für unsere Modelle zur Verfügung.

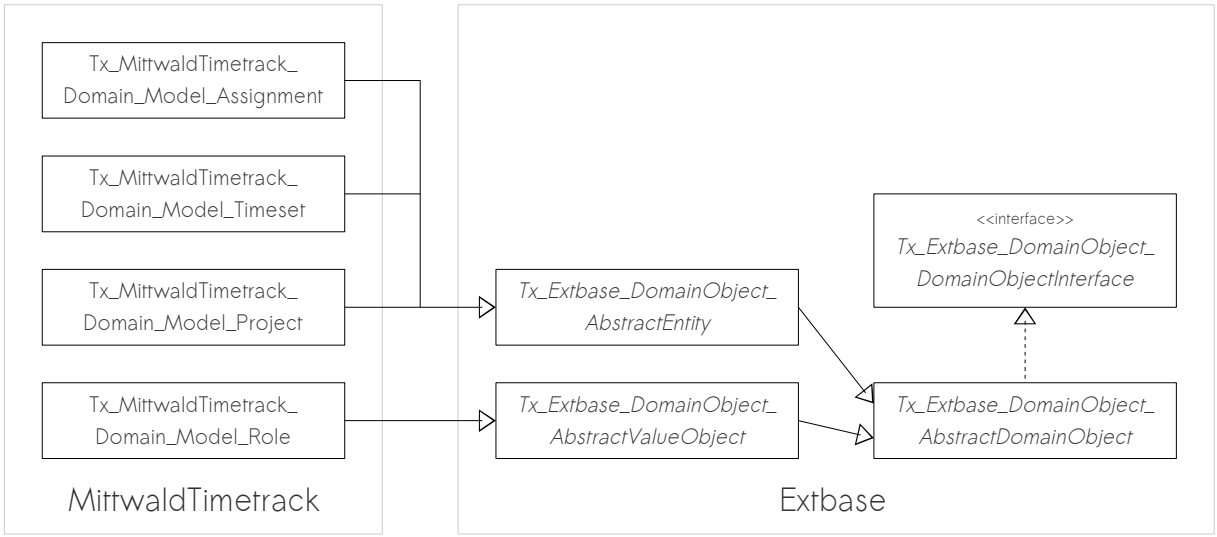

Abbildung 20: Vererbungsschema der Domänenklassen

Für die objektrelationale Abbildung ist es des weiteren wichtig, dass für sämtliche Spalten, die aus der Datenbank gelesen werden sollen, ein Klassenattribut vorhanden ist. Das Attribut muss per Konvention denselben Namen tragen wie die Datenbankspalte, die abgebildet werden soll.[10] Um mehr brauchen wir uns diesbezüglich auch nicht zu kümmern; diese Arbeit nimmt uns Extbase ab. Siehe hierzu auch den Abschnitt III.1.4.1 über das Data-Mapping auf Seite 49.

III.4.1.1.a Projekte: Die Klasse „Project“

Beginnen wir nun mit der Implementierung unseres *Project*-Objektes. Wir wir uns erinnern, zeichnet sich dieses durch einen Namen, das übergeordnete Projekt, sowie ein Anfangs- und ein Enddatum aus. Des weiteren besteht eine Assoziation mit der *Assignment*-Klasse.

Wenn zum Anlegen der Extension der Kickstarter verwendet wurde, enthält die Klasse bereits alle dort definierten Attribute und die zugehörigen sondierenden und manipulierenden Methoden. Im oberen Teil der Klasse werden zunächst sämtliche Attribute definiert:

10 Einzige Ausnahme: Enthält der Name der Datenbankspalte Unterstriche, kann für das entsprechende Klassenattribut stattdessen der *lowerCamelCase* verwendet werden. Eine Datenbankspalte `hello_world` wird also auch auf das Klassenattribut `$helloWorld` abgebildet.

PHP: Classes/Domain/Model/Project.php

```
<?php

class Tx_MittwaldTimetrack_Domain_Model_Project↲
    extends Tx_Extbase_DomainObject_AbstractEntity {

        /**
         * The project name
         * @var string
         * @validate NotEmpty
         */
    protected $name;

        /**
         * The project start time
         * @var integer
         * @validate NotEmpty
         */
    protected $start;

        /**
         * The project end time
         * @var integer
         * @validate NotEmpty
         */
    protected $end;

        /**
         * assignments
         * @var Tx_Extbase_Persistence_ObjectStorage↲
                    <Tx_MittwaldTimetrack_Domain_Model_Assignment>
         * @lazy
         * @cascade delete
         */
    protected $assignments;

        /**
         * children
         * @var Tx_Extbase_Persistence_ObjectStorage↲
                    <Tx_MittwaldTimetrack_Domain_Model_Project>
         */
    protected $children;

        /**
         * A timetracking project
         * @var Tx_MittwaldTimetrack_Domain_Model_Project
         */
```

```
    protected $project;

[...]
```

Zunächst enthält unsere Klasse schon einmal alle Attribute, die wir zuvor in dem Domänenmodell festgelegt hatten. Außerdem hinzugekommen ist das Klassenattribut `$assignments`, als Relation zur *Assignment*-Klasse. Bei diesem handelt es sich um ein Objekt der Klasse *Tx_Extbase_Persistence_ObjectStorage*. Diese wird – ähnlich einem Array – zur Speicherung mehrerer Objekte verwendet. Über dieses Objekt werden wir also später auf alle Mitgliedschaften zugreifen können, die dem jeweiligen Projekt zugeordnet sind.

An diesem Beispiel sehen wir auch sehr gut, wie Extbase Assoziationen mit unterschiedlichen Multiplizitäten behandelt. Eine Assoziation mit der Multiplizität 1 – schließlich kann jedes Projekt nur genau ein übergeordnetes Projekt haben – wird direkt als Referenz auf eine Instanz der entsprechenden Klasse abgebildet. Im Gegensatz dazu wird eine Assoziation mit höherer Multiplizität als eine Instanz der Klasse *Tx_Extbase_Persistence_ObjectStorage*[11] abgebildet, welche alle assoziierten Objekte enthält.

> **Hinweis:** Die Kommentare über den Attributen sind wichtig und sollten auf jeden Fall angegeben werden. Extbase wertet diese aus und führt eine Validierung durch, wenn eines dieser Attribute neu gesetzt wird. Siehe hierzu auch den Abschnitt III.1.4.2 über Reflexion auf Seite 50.
>
> Bisher nicht erläutert hatten wir die Anmerkungen `@lazy` und `@cascade delete` über dem assignments-Attribut (diese kommen nicht vom Kickstarter, sondern müssen manuell hinzugefügt werden; es schadet aber auch nicht direkt, sie nicht anzugeben). Die `@lazy`-Anweisung bedeutet, dass das Objekt erst dann aus der Datenbank geladen werden soll, wenn es tatsächlich benötigt wird. `@cascade delete` bedeutet, dass die zugeordneten Objekte gelöscht werden sollen, wenn das Projekt gelöscht wird. Siehe hierzu mehr im Kapitel IV.4.3 über das *Lazy Loading* auf Seite 141.

Als nächstes folgen nun der Konstruktur, der die leeren Objekt-Storages initialisiert, sowie einige sondierende und manipulierende Methoden für die Attribute:

PHP: Classes/Domain/Model/Project.php

```
    /**
     * Constructor. Initializes all Tx_Extbase_Persistence_ObjectStorage
instances.
```

11 Bei dieser Klasse handelt es sich um eine Datenstruktur ähnlich einem Array. Sie stellt dieselben Schnittstellen bereit wie die *SplObjectStorage*-Klasse der *Standard PHP Library*, die ab PHP 5.3.0 standardmäßig enthalten ist, und in FLOW3 auch anstelle dieser Klasse verwendet wird: http://de2.php.net/manual/en/class.splobjectstorage.php

```
     */
public function __construct() {
    $this->assignments = new Tx_Extbase_Persistence_ObjectStorage();
    $this->children = new Tx_Extbase_Persistence_ObjectStorage();
}

    /**
     * Setter for name
     * @param string $name The project name
     * @return void
     */
public function setName($name) {
    $this->name = $name;
}

    /**
     * Getter for name
     * @return string The project name
     */
public function getName() {
    return $this->name;
}

[...]
```

Eine Besonderheit ist nun noch der Zugriff auf das Attribut `$assignments`. Während eine sondierende Methode *getAssignments*, die alle Zuweisungen für dieses Projekt ermittelt, noch durchaus Sinn macht, wäre es nur begrenzt sinnvoll, eine Methode *setAssignments* zur Verfügung zu stellen.[12] Praktischer wäre es, dem Projekt einzelne Mitgliedschaften zuordnen zu können. Der Kickstarter implementiert aus diesem Grund zusätzlich zu *get-* und *setAssignments* auch noch die Methoden *addAssignment* und *removeAssignment*, mit welchem einzelne Mitgliedschaften hinzugefügt oder entfernt werden können:

PHP: Classes/Domain/Model/Project.php

```
    /**
     * Setter for assignments
     * @param Tx_Extbase_Persistence_ObjectStorage ↲
         <Tx_MittwaldTimetrack_Domain_Model_Assignment> $assignments assignments
     * @return void
     */
public function setAssignments ↲
  (Tx_Extbase_Persistence_ObjectStorage $assignments) {
```

12 Wenn der Kickstarter zum Erstellen der Extension eingesetzt wird, würde der in diesem Fall die Methode *setAssignments* – zusammen mit *add-* und *removeAssignment* – trotzdem anlegen.

```
    $this->assignments = $assignments;
}

    /**
     * Getter for assignments
     * @return Tx_Extbase_Persistence_ObjectStorage ↲
                <Tx_MittwaldTimetrack_Domain_Model_Assignment> assignments
     */
public function getAssignments() {
    return $this->assignments;
}

    /**
     * Adds an Assignment
     * @param Tx_MittwaldTimetrack_Domain_Model_Assignment ↲
               The Assignment to be added
     * @return void
     */
public function addAssignment ↲
  (Tx_MittwaldTimetrack_Domain_Model_Assignment $assignment) {
    $this->assignments->attach($assignment);
}

    /**
     * Removes an Assignment
     * @param Tx_MittwaldTimetrack_Domain_Model_Assignment ↲
               The Assignment to be removed
     * @return void
     */
public function removeAssignment↲
  (Tx_MittwaldTimetrack_Domain_Model_Assignment $assignment) {
    $this->assignments->detach($assignment);
}
```

Die Klasse *Tx_Extbase_Persistence_ObjectStorage* bietet also die Methoden *attach* und *detach* an, um der Liste einzelne Objekte hinzuzufügen oder wieder zu entfernen. Möchten wir alle Zuweisungen löschen, erstellen wir einfach eine neue – leere – Liste.

Zusätzlich zu den standardmäßig vom Kickstarter angelegten Methoden können natürlich beliebige weitere Methoden hinzugefügt werden. So zum Beispiel, um ohne den Umweg über ein *Assignment*-Objekt direkt einen Benutzer dem Projekt zuweisen zu können, oder anders herum zu ermitteln, ob ein bestimmter Benutzer dem Projekt zugeordnet ist:

PHP: Classes/Domain/Model/Project.php

```
    /**
     * Adds a user to this project.
     * @param Tx_Extbase_Domain_Model_FrontendUser $user ↲
               The user to be added
     * @param Tx_MittwaldTimetrack_Domain_Model_Role $role ↲
               The role for the new user
     * @return void
     */
public function assignUser ( Tx_Extbase_Domain_Model_FrontendUser   $user,
                             Tx_MittwaldTimetrack_Domain_Model_Role $role ) {
    $assignment = New Tx_MittwaldTimetrack_Domain_Model_Assignment();
    $assignment->setUser($user);
    $assignment->setRole($role);
    $this->addAssignment($assignment);
}

    /**
     * Gets an assignment for a specific user.
     * @param Tx_Extbase_Domain_Model_FrontendUser $user ↲
               The user whose assignment is to be loaded
     */
public function getAssignmentForUser ↲
  ( Tx_Extbase_Domain_Model_FrontendUser $user ) {
    foreach ( $this->getAssignents() as &$assignment)
        if($assignment->getUser()->getUid() === $user->getUid())
            return $assignment;
    return NULL;
}

    /**
     * Determines if a user is assigned to this project.
     * @param Tx_Extbase_Domain_Model_FrontendUser $user The user
     * @return boolean TRUE, if the user is a project member, otherwise FALSE.
     */
public function isAssigned ( Tx_Extbase_Domain_Model_FrontendUser $user ) {
    return $this->getAssignmentForUser($user) !== NULL;
}
```

Wir werden diese Klasse später noch um einige Methoden erweitern. Doch wenden wir uns zunächst einmal den anderen Bestandteilen des Models zu.

III.4.1.1.b Benutzerrollen: Die Klasse „Role“

Bei dem *Role*-Objekt handelt es sich um ein Wertobjekt, welches lediglich durch seinen Namen ausgezeichnet wird. Die Implementierung der Klasse *Role* fällt daher nicht sonderlich komplex aus. Da wir uns an dieser Stelle damit begnügen wollen, die Benutzerrollen über das TYPO3-Backend anzulegen, bräuchten wir die Methode *setName* eigentlich noch nicht einmal.

PHP: Classes/Domain/Model/Role.php

```
class Tx_MittwaldTimetrack_Domain_Model_Role ↲
  extends Tx_Extbase_DomainObject_AbstractValueObject {

        /**
         * The role name
         * @var string
         * @validate NotEmpty
         */
    protected $name;

        /**
         * Setter for name
         * @param string $name The role name
         * @return void
         */
    public function setName($name) {
        $this->name = $name;
    }

        /**
         * Getter for name
         * @return string The role name
         */
    public function getName() {
        return $this->name;
    }

}
```

III.4.1.1.c Benutzer: Die Klasse „FrontendUser“

Darum, eine Klasse für die Abbildung der Benutzer zu entwickeln, kommen wir zum Glück herum. Extbase stellt standardmäßig bereits eine Klasse sowohl für Frontendbenutzer als auch für Frontendgruppen sowie zwei entsprechende Repositories zur Verfügung. Diese Klassen können problemlos in unserer Erweiterung verwendet werden.

Die Klasse *Tx_Extbase_Domain_Model_FrontendUser* bietet sondierende Methoden für nahezu alle Attribute der *fe_users*-Tabelle an, so zum Beispiel eine Methode *getUsername* oder *getEmail*.

III.4.1.1.d Zuweisungen: Die Klasse „Assignment“

Die Klasse *Assignments* wird benötigt, um jeder Assoziation zwischen einem Benutzer und einem Projekt eine Rolle zuweisen zu können. Eine solche Assoziation zeichnet sich also durch einen Benutzer, ein Projekt und eine Rolle aus:

PHP: Classes/Domain/Model/Assignment.php

```
class Tx_MittwaldTimetrack_Domain_Model_Assignment ↲
  extends Tx_Extbase_DomainObject_AbstractEntity {

        /**
         * role
         * @var Tx_MittwaldTimetrack_Domain_Model_Role
         */
    protected $role;

        /**
         * timesets
         * @var Tx_Extbase_Persistence_ObjectStorage
         */
    protected $timesets;

        /**
         * project
         * @var Tx_MittwaldTimetrack_Domain_Model_Project
         */
    protected $project;

    ...
```

Für alle Attribute hat der Kickstarter auch gleich die passenden sondierenden und manipulierenden Methoden erstellt. Es fehlt jedoch noch die Assoziation zur *fe_users*-Tabelle, da wir diese über den Kickstarter zuvor nicht abbilden konnten. Daher muss das entsprechende Attribut mit den dazugehörigen Methoden nun noch erstellt werden:

PHP: Classes/Domain/Model/Assignment.php

```
        /**
         * fe_user
         * @var Tx_Extbase_Domain_Model_FrontendUser
         */
    protected $user;
```

```
...

    /**
     * Returns the associated user
     * @return Tx_Extbase_Domain_Model_FrontendUser The associated user
     */
public function getUser() {
    return $this->user;
}

    /**
     * Sets the associated user
     * @param Tx_Extbase_Domain_Model_FrontendUser $user The assiciated user
     * @return void
     */
public function setUser(Tx_Extbase_Domain_Model_FrontendUser $user) {
    $this->user = $user;
}
```

Eine vollständige Implementation der Klasse finden Sie in dem Extension-Paket, welches zusammen mit diesem Dokument zum Download angeboten wird.[13]

III.4.1.1.e Zeitpaare: Die Klasse „Timeset“

Wie bereits besprochen, zeichnet sich die Klasse *Timeset* durch eine Anfangs- und eine Stoppzeit aus und ist mit einer Benutzerzuweisung assoziiert. Wir können die Klasse *Tx_MittwaldTimetrack_Domain_Model_Timeset* also wie gewohnt mit diesen Attributen erstellen:

PHP: Classes/Domain/Model/Timeset.php

```
class Tx_MittwaldTimetrack_Domain_Model_Timeset ↲
  extends Tx_Extbase_DomainObject_AbstractEntity {

    /**
     * The assignment
     * @var Tx_MittwaldTimetrack_Domain_Model_Assignment
     */
  protected $assignment;

    /**
     * The start time of this timeset
     * @var DateTime
     */
  protected $starttime;
```

13 http://www.mittwald.de/extbase-dokumentation/

```
        /**
         * The stop time of this timeset
         * @var DateTime
         */
    protected $stoptime;

        /**
         * A comment for this timeset
         * @var string
         */
    protected $comment;

    ...
```

Zusätzlich zu den üblichen Getter- und Settermethoden können wir nun auch noch beispielsweise eine Methode implementieren, welche die tatsächlich gearbeitete Zeit aus der Start- und der Stoppzeit ermittelt:

PHP: Classes/Domain/Model/Timeset.php

```
public function getWorkedTime() {
    return intval($this->stoptime->format('U'))
      - intval($this->starttime->format('U'));
}
```

`format('U')` liefert das angegebene Datum als UNIX-Zeitstempel. Ab PHP 5.3 steht hierfür auch die Methode *DateTime::getTimestamp()* zur Verfügung.

III.4.1.1.f Wie alle Klassen zusammenarbeiten

Die meisten Leute kennen den Begriff des *Delegierens* wahrscheinlich nur aus dem Personalwesen. Tatsächlich aber können wir von diesem Prinzip auch in der Softwareentwicklung gut Gebrauch machen.

Führen wir uns einmal folgendes vor Augen: Wir können pro Mitarbeiterzuweisung alle Zeitpaare ermitteln, die der entsprechende Mitarbeiter für das Projekt gebucht hat. Pro Zeitpaar können wir wiederum ermitteln, wie lange eigentlich gearbeitet wurde (indem wir Start- und Stoppzeit voneinander subtrahieren). Ein naheliegender Gedanke wäre nun, direkt für jede Mitarbeiterzuweisung die gearbeitete Zeit zu ermitteln („Wie lange hat Mitarbeiter X insgesamt an dem Projekt Y gearbeitet?“). Da wir pro Mitarbeiter alle Zeitpaare und pro Zeitpaar die gearbeitete Zeit ermitteln können, stellt dies kein großes Problem mehr dar. Wir müssen nur die Methoden, die wir bereits haben, kombinieren. Erweitern wir also unsere *Assignment*-Klasse um folgende Methode:

PHP: Classes/Domain/Model/Assignment.php

```
public function getWorkedTime() {
    $time = 0;
    foreach($this->getTimesets() as $timeset)
        $time += $timeset->getWorkedTime();
    return $time;
}
```

Dieses Prinzip lässt sich nun noch weiter fortführen: Schließlich können wir auch pro Projekt alle Mitarbeiterzuweisungen ermitteln. Wie wäre es nun, wenn wir pro Projekt ermitteln könnten, wie lange alle Mitarbeiter zusammen daran gearbeitet haben?

Fügen wir also auch in die *Project*-Klasse eine Methode *getWorkedTime* ein:

PHP: Classes/Domain/Model/Project.php

```
public function getWorkedTime() {
    $time = 0;
    foreach($this->getAssignments() as $assignment)
        $time += $assignment->getWorkedTime();
    return $time;
}
```

Dieses Prinzip wird als *Delegieren* bezeichnet, da jede dieser beiden Methoden lediglich eine Methode einer untergeordneten Klasse aufruft und die Ergebnisse aufsummiert.

III.4.1.2 Repositories

Da die Objekte *Project* und *Role* jeweils global zugänglich sein müssen, brauchen wir nun jeweils eine Repository-Klasse für jedes dieser beiden Objekte. Extbase stellt hierfür die abstrakte Basisklasse *Tx_Extbase_Persistence_Repository* zur Verfügung. Diese sollten wir auch auf jeden Fall verwenden, da sie uns jede Menge Arbeit abnimmt.

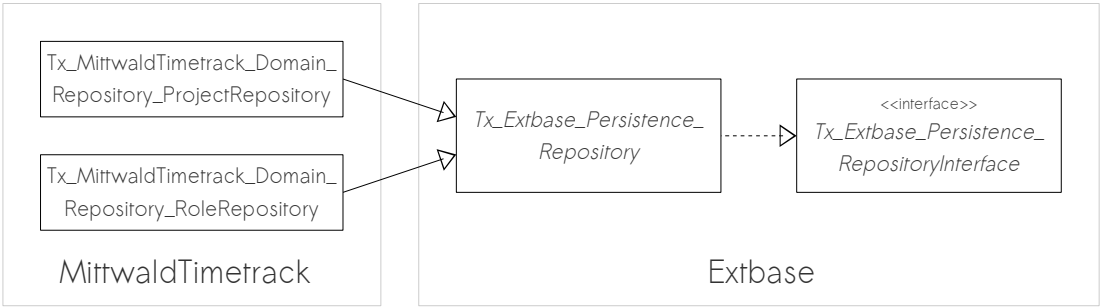

Abbildung 21: Vererbungsdiagramm der Repository-Klassen

Die Repository-Klassen werden im Verzeichnis *Classes/Domain/Repository* gespeichert. Per Konvention müssen die Klassennamen mit „Repository“ enden. Unsere Klassen heißen also:

- *Tx_MittwaldTimetrack_Domain_Repository_ProjectRepository* und
- *Tx_MittwaldTimetrack_Domain_Repository_RoleRepository*

Beginnen wir also mit der Implementierung unserer Repositories:

PHP: Classes/Domain/Repository/ProjectRepository.php

```
class Tx_MittwaldTimetrack_Domain_Repository_ProjectRepository ↲
    extends Tx_Extbase_Persistence_Repository {
}
```

PHP: Classes/Domain/Repository/RoleRepository.php

```
class Tx_MittwaldTimetrack_Domain_Repository_RoleRepository ↲
    extends Tx_Extbase_Persistence_Repository {
}
```

Dies mag zunächst überraschen – aber mehr ist im Moment tatsächlich nicht nötig, damit unsere Repositories funktionieren. Beide Klassen erben die komplette Programmlogik aus der Basisklasse; diese ist imstande, aus dem Klassennamen des Repositories das dazugehörige Modell zu ermitteln (also zum Beispiel *Tx_MittwaldTimetrack_Domain_Model_Project* für das *ProjectRepository*). Anhand dieser Information ist das Repository ohne weiteres Zutun in der Lage, sämtliche Attribute des Domänenobjektes zu durchsuchen. Hierzu stellt das Repository folgende Methoden zur Verfügung:

Methode	Beschreibung
findAll()	Liefert alle Objekte, die nicht gelöscht sind.
findByUid($uid)	Liefert ein einzelnes Objekt mit einer bestimmten UID.
findBy[Attribut] ($v)	Liefert alle Objekte, bei denen das durch *[Attribut]* angegebene Attribut mit `$v` übereinstimmt. So könnten mit folgendem Beispiel etwa alle Projekte mit einem bestimmten Namen ermittelt werden: `$projects = $repository->findByName("Foo");` `ForEach($projects As $project)` `  Echo $project->getName().' ';`

findOneBy[Attribut] ($v)	Liefert das erstbeste Objekt, bei dem die durch [Attribut] angegebene Eigenschaft mit $v übereinstimmt. Beispiel:
	`$project = $repository->findOneByName("Bar");`
	`If($project == NULL) echo "Nicht gefunden";`
	`Else Echo $project->getName();`

Für komplexere Anwendungen reichen diese Standardmethoden zwar nicht aus, für den Anfang kommen wir jedoch ganz gut damit zurecht. Für kompliziertere Abfragen können jederzeit eigene Methoden implementiert werden. Dies werden wir in Abschnitt IV.3 auf Seite 130 näher betrachten.

Das Repository dient nicht nur dazu, Objekte aus der Datenbank auszulesen, sondern auch genauso dazu, Objekte wieder in der Datenbank zu speichern. Hierzu stellt jedes Repository die Methoden *add*, *remove* und *update* zur Verfügung:

Methode	Beschreibung
add ($o)	Fügt das Objekt $o als neues Element in die Datenbank ein. Das Objekt darf als solches noch nicht persistent gespeichert worden sein. Ist das neue Objekt die Wurzel eine Aggregates, so werden die anderen Bestandteile des Aggregates – soweit vorhanden – ebenfalls persistiert.

update ($o)	Speichert Änderungen, die an dem Objekt vorgenommen wurden, in der Datenbank. Hierzu muss das Objekt bereits existieren.
remove ($o)	Entfernt ein Objekt wieder aus der Datenbank. Handelt es sich bei dem Objekt um eine Aggregatwurzel, so werden die untergeordneten Bestandteile des Aggregates ebenfalls gelöscht.

Verwendet werden könnte ein solches Repository beispielsweise wie folgt:

```
$projectRepository =&
  t3lib_div::makeInstance('Tx_MittwaldTimetrack_Domain_Repository_ProjectReposit
ory');

  // Speichern eines neuen Objektes
$projectRepository->add($project);

  // Bearbeiten eines Objektes
$projectRepository->update($project);

  // Löschen eines Objektes
$projectRepository->remove($project);
```

III.4.2 Der erste Controller

Erstellen wir nun unseren ersten Controller, den ProjectController. Die Controller werden grundsätzlich unter *Classes/Controller/* gespeichert; die Namen müssen mit „Controller“ enden. Wie eigentlich für alles stellt Extbase auch für einen Action-Controller eine abstrakte Basisklasse bereit. Diese heißt *Tx_Extbase_MVC_Controller_ActionController*.

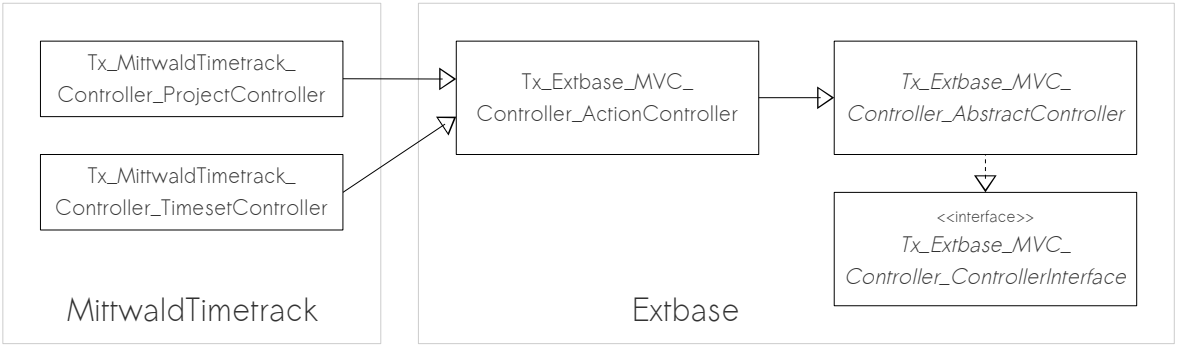

Abbildung 22: Vererbungsdiagramm der Controller-Klassen

Beginnen wir zunächst mit der *index*-Action, die eine Liste sämtlicher zur Verfügung stehender Projekte anzeigen soll. Per Konvention wird jede Aktion als öffentliche Methode implementiert, die dem Namensschema *[Aktionsname]Action* folgt. Unser *ProjectController* könnte also aussehen wie folgt:

PHP: Classes/Controller/ProjectController.php

```
class Tx_MittwaldTimetrack_Controller_ProjectController ↲
    extends Tx_Extbase_MVC_Controller_ActionController {

        /**
         * @var Tx_MittwaldTimetrack_Domain_Repository_ProjectRepository
         */
    protected $projectRepository;

        /**
         * Initializes the current action
         * @return void
         */
    protected function initializeAction() {
        $this->projectRepository =& t3lib_div::makeInstance ( ↲
          "Tx_MittwaldTimetrack_Domain_Repository_ProjectRepository" );
    }

        /**
         * List action for this controller. Displays all projects.
         */
    public function indexAction() {
        $projects = $this->projectRepository->findAll();
        $this->view->assign('projects', $projects);
    }

        /**
         * Action that displays a single Project
         * @param Tx_MittwaldTimetrack_Domain_Model_Project $project ↲
                  The Project to display
         */
    public function showAction ↲
      (Tx_MittwaldTimetrack_Domain_Model_Project $project) {
        $this->view->assign('project', $project);
    }

        /**
         * Displays a form for creating a new Project
         *
         * @param Tx_MittwaldTimetrack_Domain_Model_Project $newProject ↲
             A fresh Project object taken as a basis for the rendering
         * @dontvalidate $newProject
         */
```

```
public function newAction ( ↲
  Tx_MittwaldTimetrack_Domain_Model_Project $newProject = NULL ) {
    $this->view->assign('newProject', $newProject);
}

    /**
     * Creates a new Project and forwards to the index action.
     *
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $newProject ↲
         A fresh Project object which has not yet been added to ↲
         the repository
     */
public function createAction ↲
  (Tx_MittwaldTimetrack_Domain_Model_Project $newProject) {
    $this->projectRepository->add($newProject);
    $this->flashMessageContainer->add('Your new Project was created.');
    $this->redirect('index');
}

    /**
     * Displays a form to edit an existing Project
     *
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project ↲
         The Project to display
     * @dontvalidate $project
     */
public function editAction ↲
 (Tx_MittwaldTimetrack_Domain_Model_Project $project) {
    $this->view->assign('project', $project);
}

    /**
     * Updates an existing Project and forwards to the index action.
     *
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project ↲
         The Project to display
     */
public function updateAction ↲
 (Tx_MittwaldTimetrack_Domain_Model_Project $project) {
    $this->projectRepository->update($project);
    $this->flashMessageContainer->add('Your Project was updated.');
    $this->redirect('index');
}

    /**
     * Deletes an existing Project
     *
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project ↲
          The Project to be deleted
```

```
         */
    public function deleteAction ↲
      (Tx_MittwaldTimetrack_Domain_Model_Project $project) {
        $this->projectRepository->remove($project);
        $this->flashMessageContainer->add('Your Project was removed.');
        $this->redirect('index');
    }

}
```

Dieser Quelltext ist ja zunächst einmal noch recht überschaubar. Zur Erläuterung: Die Methode *initializeAction()* wird vor jeder anderen Action des Controllers aufgerufen. Auf diese Weise steht in jeder Aktion bereits eine Instanz des Projekt-Repositories zur Verfügung.

In der Index-Aktion schließlich bemühen wir das Repository, um alle zur Verfügung stehenden Projekte aus der Datenbank zu laden. Diese werden dann an den View weitergereicht, welcher sich um die Darstellung kümmert. Die Initialisierung des Views ist übrigens bereits in der Basisklasse verankert, sodass wir uns nicht weiter darum kümmern müssen.

III.4.3 Darstellung von Daten

III.4.3.1 Der erste View

Nachdem wir die Domäne und einen ersten Controller auf die Beine gestellt haben, können wir uns nun darum kümmern, dass unsere Daten auch angezeigt werden. Standardmäßig ist jeder Controller-Aktion genau ein View zugeordnet. Die HTML-Vorlage für diesen View wird unter dem Dateinamen *Resources/Private/Templates/[Controllername]/[Aktionsname].html* gesucht. Die Designvorlage für unsere Index-Aktion wird also unter dem Dateinamen *Resources/Private/Templates/Project/index.html* gespeichert.

Wenn für das Erstellen der Erweiterung der Kickstarter verwendet wurde, war dieser bereits mal wieder fleißig und hat für alle Controller-Aktionen bereits Standard-Vorlagen eingerichtet. Um uns schrittweise an das Problem heranzutasten, beginnen wir jedoch zunächst einmal mit folgendem View:

HTML: Resources/Private/Templates/Project/index.html

```
<h2>Projekte</h2>

<table>
  <tr>
```

```
      <th>Projektname</th>
      <th>Anzahl Mitarbeiter</th>
      <th>Aufgewendete Zeit</th>
    </tr>
    <f:if condition="{projects}">
      <f:then>
        <f:for each="{projects}" as="project">
          <tr>
            <td>{project.name}</td>
            <td><f:count subject="{project.assignments}" /> Mitarbeiter</td>
            <td>{project.workedTime}<td>
          </tr>
        </f:for>
      </f:then><f:else>
        <tr><td>Keine Projekte gefunden</td></tr>
      </f:else>
    </f:if>
  </table>
```

Da wir nun endlich unseren ersten View erstellt haben, können wir endlich im Frontend überprüfen, ob die neue Erweiterung überhaupt so funktioniert wie gedacht. Wählen wir also die Seite mit dem Plugin im Frontend an, und wir erhalten – hoffentlich – folgende Ausgabe:

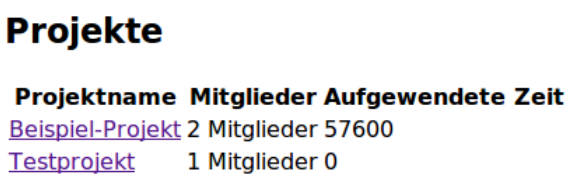

Abbildung 23: Die index-View des Projekt-Controllers

Diese Ausgabe hat noch ein paar Schönheitsfehler – zum Beispiel wird die aufgewendete Zeit in Sekunden angezeigt; hier wären vielleicht Stunden oder sogar Tage besser – und auch die optische Aufmachung könnte schöner sein. Doch darum kümmern wir uns später; immerhin funktioniert unsere Erweiterung schon einmal!

III.4.3.2 Mehrsprachigkeit

Eine gute Lokalisierbarkeit gehört zu den Grundanforderungen jeder TYPO3-Erweiterung. Sprachabhängige Texte direkt in die HTML-Vorlage zu schreiben – wie wir es gerade im letzten Abschnitt gemacht haben – wirkt dem natürlich eher entgegen. Sinnvoller ist es, diese Texte in eine separate Datei auszulagern, aus der sie in Abhängigkeit der jeweiligen Frontend-Sprache entsprechend ausgelesen werden können. Dies ist üblicherweise die Datei *locallang.xml*. Bei Extbase-Erweiterungen wird diese im Verzeichnis *Resources/Private/Language/* gespeichert.

Das Format der *locallang.xml* hat sich im Vergleich zu bisherigen TYPO3-Erweiterungen nicht verändert. Der Zugriff auf die darin enthaltenen Texte kann in Fluid durch die Verwendung des *TranslateViewHelpers* erfolgen (dieser wurde in Abschnitt III.2.3 auf Seite 54 bereits angesprochen). Dieser ViewHelper kann wie folgt angesprochen werden:

```
<f:translate key="project_index_title" />
```

Alternativ kann auch die nachfolgende Inline-Notation verwendet werden. Diese funktioniert zum Beispiel auch, wenn sie als Attribut an einen anderen ViewHelper übergeben wird:

```
{f:translate(key:'project_index_title')}

    <!-- Beispiel für Verschachtelte ViewHelper: Erstelle einen Button zum
         Speichern eines Formulares, mit einer sprachabhängigen Beschriftung.
         form.submit ist selbst ein ViewHelper und bekommt einen anderen
         ViewHelper (translate) als Parameter übergeben. //-->
<f:form.submit value="{f:translate(key:'submit_button')}" />
```

III.4.4 Kommunikation mit dem Controller: Eine Detailansicht

III.4.4.1 Entgegennahme von Parametern

Nachdem wir eine Übersichtsliste über alle Projekte haben, können wir im nächsten Schritt eine Ansicht implementieren, in der die Details zu einem ganz bestimmten Projekt dargestellt werden. Hierzu benötigen wir eine weitere Controller-Aktion, die den Namen *show* tragen soll.

Hier stehen wir auch schon gleich vor der nächsten Herausforderung: Damit der Controller die Details zu einem Projekt anzeigen kann, muss er zunächst einmal wissen, um welches Projekt es überhaupt geht. Dies bedeutet, dass wir irgendeine Möglichkeit brauchen, um dem Controller weitere Parameter mitzuteilen. In Extbase ist dies relativ einfach: Wir fügen der entsprechenden Methode im Controller einfach einen Parameter hinzu:

PHP: Classes/Controller/ProjectController.php

```
    /**
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project
     */
public function showAction ( Tx_MittwaldTimetrack_Domain_Model_Project $project)
{
    $this->view->assign('project', $project);
}
```

Wird dieser Controller nun aufgerufen, wertet Extbase automatisch die GET- und POST-Parameter aus und weist diese den entsprechenden Methodenparametern zu. Es können auch komplette Domänenobjekte als Parameter entgegen genommen werden, wie in diesem Fall das darzustellende Projekt.

Des weiteren ist gerade hier die Verwendung von DocComment-Kommentaren sehr wichtig. Parameter, die nicht im Kommentar dokumentiert sind oder einen falschen Datentyp aufweisen, quittiert Extbase gnadenlos mit einer Fehlermeldung. Außerdem wichtig ist die Benennung der Parameter. So muss der Methodenparameter genau denselben Namen tragen wie der GET- oder POST-Parameter, damit eine korrekte Zuordnung erfolgen kann.

III.4.4.2 Verlinkung mit zusätzlichen Argumenten

Das nächste Ziel soll es sein, aus der Übersichtsansicht heraus auf die Detailansicht zu verlinken. Hierzu werden wir uns des *ActionLink*-ViewHelpers bedienen. Bauen wir also unser Übersichtstemplate wie folgt um:

```
<!-- ... -->
<f:for each="{projects}" as="project">
  <tr>
    <td>
      <f:link.action action="show" arguments="{project : project}">
        {project.name}
      </f:link.action>
    </td><td>
      <f:count subject="{project.assignments}" /> Mitarbeiter
    </td><td>
```

```
        {project.workedTime}
      <td>
    </tr>
  </f:for>
  <!-- ... -->
```

Mit dem *arguments*-Attribut dieses ViewHelpers können wir zusätzliche Argumente angeben, welche in dem Link als GET-Parameter eingefügt werden sollen. Diese werden als assoziatives Array angegeben, die Notation orientiert sich dabei an dem JSON Standard (Array in geschweiften Klammern, Index und Wert durch einen Doppelpunkt getrennt). Es ist kein Problem, auch komplexe Domänenobjekte als Parameter zu übergeben; beim Erstellen der URL wird lediglich die ID des Objektes verwendet; der verlinkte Controller bildet diese ID dann wieder auf ein Objekt ab.

III.4.4.3 Die nächste View

Für die Detailansicht möchten wir eine Liste mit Informationen über das ausgewählte Projekt (also zum Beispiel das Start- und Enddatum), sowie – sofern vorhanden – jeweils eine Liste mit allen Unterprojekten sowie allen zugeordneten Mitarbeitern und den gearbeiteten Zeiten:

HTML: Resources/Private/Templates/Project/show.html

```
<h2>Projekt-Details: {project.name}</h2>

<div>
  Start: <f:format.date format="d. m. Y">{project.start}</f:format.date>
</div>
<div>
  Ende: <f:format.date format="d. m. Y">{project.end}</f:format.date>
</div>

<h3>Unterprojekte</h3>

<table>
  <tr>
    <th>Projektname</th>
    <th>Mitarbeiter</th>
    <th>Aufgewendete Zeit</th>
  </tr>
  <f:for each="{project.children}" as="subproject">
    <tr>
      <td>
        <f:link.action action="show" arguments="{project : subproject}">
          {subproject.name}
        </f:link>
      </td>
```

```
      <td>
        <f:count subject="{subproject.assignments}" /> Mitarbeiter
      </td><td>
        {subproject.workedTime}
      <td>
    </tr>
  </f:for>
</table>

<h3>Mitarbeiter</h3>

<table>
  <tr>
    <th>Mitarbeiter</th>
    <th>Rolle</th>
    <th>Aufgewendete Zeit</th>
  </tr>
  <f:for each="{project.assignments}" as="assignment">
    <tr>
      <td>{assignment.user.name}</td>
      <td>{assignment.role.name}</td>
      <td>{assignment.workedTime}</td>
    </tr>
  </f:for>
</table>
```

Betrachten wir unser Plugin nun erneut im Frontend und wählen einen der frisch hinzugefügten Links, so erhalten wir nun im Idealfall folgende Darstellung:

Abbildung 24: Die Detail-Darstellung eines Projektes

III.4.5 Auswertung von Formulareingaben

Nachdem wir herausgefunden haben, wie unser Controller Benutzereingaben verarbeiten kann, möchten wir nun im nächsten Abschnitt betrachten, wie wir solche Eingaben persistent in Form neuer Objekte in der Datenbank speichern können. Zu diesem Zweck werden wir ein Formular implementieren, mit welchem wir neue Projekte anlegen und vorhandene Projekte bearbeiten können. Zum Abschluss dieses Abschnittes werden wir schließlich das Herz der Zeiterfassungserweiterung implementieren: Ein Formular zum Einbuchen von Zeitpaaren.

III.4.5.1 Ein Bearbeitungsformular

III.4.5.1.a Formular-ViewHelper

Fluid stellt für die Erstellung von Formularen eine ganze Gruppen von eigenen ViewHelpers zur Verfügung. Der wichtigste ist der *Form*-ViewHelper, der anstelle des `<form>`-HTML-Tags verwendet werden sollte. Dieser kann folgendermaßen verwendet werden:

```
<f:form action="create" object="{newProject}">
    Projektname: <f:form.textbox property="name" />
    <f:form.submit value="Speichern" />
</f:form>
```

Das *action*-Attribut beschreibt die Controller-Aktion, an welche die Formulareingaben weitergeleitet werden sollen. Sollte die Aktion zu einem anderen Controller gehören als die View, die das Formular darstellt, so kann auch zusätzlich ein *controller*-Attribut angegeben werden.

Der *Form*-ViewHelper kann über die folgenden Eigenschaften näher konfiguriert werden:

- *object* – Weist dem Formular ein Domänenobjekt zu, welches bearbeitet werden soll. Wenn diese Eigenschaft gesetzt ist, kann allen Eingabefeldern in diesem Formular mit der `property`-Eigenschaft ein einzelnes Attribut dieses Objekts zugewiesen werden. Im obigen Beispiel ist einem Eingabefeld das `name`-Attribut eines Projektes zugeordnet. Dieses Formularfeld wird automatisch mit dem Namen des zu bearbeitenden Projektes gefüllt. Wird das Formular abgeschickt, wird der Name des Projektes automatisch auf den hier eingegebenen Wert gesetzt.

- *name* – Der Formularname muss nur angegeben werden, wenn kein Objekt zugewiesen wurde.

Innerhalb des Formulars können bestimmte ViewHelper für die Darstellung von Eingabefeldern verwendet werden. Mit dem *Form.Textbox*-ViewHelper haben wir im obigen Beispiel bereits einen davon gesehen. Nachfolgend findet sich eine Übersicht über die wichtigsten Formular-ViewHelper. Jedem dieser Eingabefelder kann über die `property`-Eigenschaft ein Klassenattribut zugewiesen werden. Geschieht dies nicht, kann auch genau wie bei einem gewöhnlichen Eingabefeld ein `name`- und ein `value`-Attribut angegeben werden.

ViewHelper	Beschreibung	Beispiel

Form.Textbox	Stellt ein einfaches, einzeiliges Text-Eingabefeld dar.	`<f:form.textbox property="name" />`
		
		`<f:form.textbox name="customTextbox" value="Hallo Welt!" />`

Form.Textarea	Stellt ein mehrzeiliges Eingabefeld dar. Die Attribute `rows` und `cols` müssen zwingend angegeben werden.	`<f:form.textarea property="motto" rows="5" cols="80" />`

Form.Checkbox	Stellt eine Checkbox dar. Falls das Feld nicht an ein Klassenattribut gebunden ist, kann hier genau wie bei einer gewöhnlichen Checkbox die `checked`-Eigenschaft verwendet werden.	`<f:form.checkbox property="hidden" />`
		
		`<f:form.checkbox name="customCheckbox" value="1" checked="{test}==1" />`

Form.Hidden	Dieser ViewHelper dient zur Darstellung eines versteckten Formularfeldes.	`<f:form.hidden property="name" />`
		
		`<f:form.hidden name="customHidden" value="123" />`

Form.Radio	Stellt einen Radio-Button dar. Genau wie beim *Checkbox*-ViewHelper kann hier auch gegebenenfalls die `checked`-Eigenschaft verwendet werden.	`<f:form.radio property="test" value="1" /> Value 1` `<f:form.radio property="test" value="2" /> Value 2`

Form.Select	Dieser ViewHelper stellt eine Mehrfach-Auswahlbox zur Verfügung. Die zur Auswahl stehenden Werte können als Array angegeben werden.
	
	Falls diesem ViewHelper ein Array von Domänenobjekten als mögliche Werte übergeben wird, kann mit dem Parameter `optionLabelField` angegeben werden, welches Klassenattribut als Beschriftung der einzelnen Elemente verwendet werden soll.
		`<f:form.select property="favoriteBeer" options="{becks:'Becks', bud:'Budweiser'}" />`
		
		`<f:form.select property="favoriteBeer" options="{beerObjectArray}" optionLabelField="name" />`

Form.Submit	Stellt einen Button dar, um das Formular abzuschicken.	`<f:form.submit value="Speichern" />`

III.4.5.1.b Die new und create-Aktionen

Grundsätzlich empfiehlt es sich, das Darstellen des Formulars und das tatsächliche Speichern der Daten auf zwei verschiedene Controller-Aktionen aufzuteilen. Daher werden wir hier die Aktion *new* für die Darstellung eines Formulars für neue Projekte verwenden, und die Aktion *create* zum Speichern eines neuen Projekts in die Datenbank:

PHP: Classes/Controller/ProjectController.php

```
    /**
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project
     * @dontvalidate $project
     */
Public Function newAction ↲
  ( Tx_MittwaldTimetrack_Domain_Model_Project $project=NULL ) {
    $this->view->assign('project', $project);
    $this->view->assign('allProjects', $this->projectRepository->findAll());
}

    /**
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project
```

```
     */
Public Function createAction ↲
  ( Tx_MittwaldTimetrack_Domain_Model_Project $project ) {
    $this->projectRepository->add($project);
    $this->redirect('index');
}
```

Wie wir sehen, akzeptiert bereits die *new*-Aktion ein Projekt-Domänenobjekt als (optionalen) Parameter. Dies hat folgende Bewandtnis: Wenn Extbase zu dem Schluss kommt, dass ein Domänenobjekt, das als Parameter übergeben ist, nicht valide ist (also wenn beispielsweise bestimmte Attribute nicht gefüllt sind), dann leitet es von der *create*-Aktion wieder auf die *new*-Aktion zurück. Wird das Projekt-Formular also nur halb ausgefüllt und abgeschickt, so wird der Benutzer schließlich wieder zum Formular zurückgeführt und es wird eine Fehlermeldung angezeigt. Dadurch, dass die *new*-Aktion nun das halbfertige Projekt als Parameter akzeptiert (die `@dontvalidate`-Anmerkung im Kommentar verhindert eine Validierung), können die Formularfelder mit den bereits ausgefüllten Attributen gefüllt werden. Das Thema der Validierung wird in Abschnitt III.4.5.2 auf Seite 96 sowie später in Abschnitt IV.5.1 auf Seite 143 vertieft.

Die *create*-Aktion erwartet schließlich ein valides (!) Domänenobjekt als Parameter. Dieses wird über die *add*-Methode des Projekt-Repositories persistent in der Datenbank gespeichert. Anschließend wird auf die *index*-Aktion des Controllers weitergeleitet.

Die View für die *new*-Aktion könnte beispielsweise aussehen wie folgt:

HTML: Resources/Private/Templates/Project/new.html

```
<h2>Neues Projekt</h2>

<f:form action="create" object="{project}">
  <table>
    <tr>
      <td>Projektname:</td>
      <td><f:form.textbox property="name" /></td>
    </tr><tr>
      <td>Übergeordnetes Projekt:</td>
      <td>
        <f:form.select property="parent"
                       options="{allProjects}"
                       optionLabelField="name" />
      </td>
    </tr>
  </table>
  <f:form.submit value="Speichern" />
</f:form>
```

Damit das neue Formular erreichbar ist, bauen wir in der *index*-View noch an einer beliebigen Stelle einen Link zur *new*-Aktion ein:

HTML: Resources/Private/Templates/Project/index.html

```
<ul>
  <li><f:link.action action="new">Neues Projekt</f:link.action></li>
</ul>
```

Folgen wir nun im Frontend diesem Link, so können wir anschließend das neue Formular betrachten:

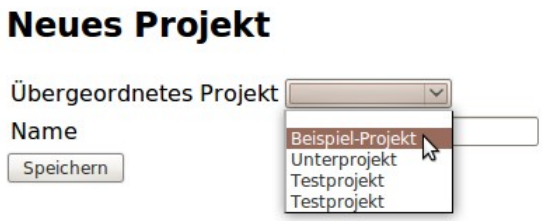

Abbildung 25: Das Formular zum Erstellen neuer Projekte

Füllen wir dieses Formular nun mit beliebigen Daten aus und wählen den Speichern-Button, so wird das neue Projekt mit den angegebenen Attributen angelegt und der Benutzer wird automatisch wieder auf die Übersichtsansicht weitergeleitet.

III.4.5.1.c Die edit und update-Aktionen

Analog zum Anlegen neuer Projekte, werden wir die *edit*-Aktion zur Darstellung eines Bearbeitungsformulars für bereits vorhandene Projekte verwenden, und die update-Aktion, um Änderungen an einem Projekt in der Datenbank zu speichern:

PHP: Classes/Controller/ProjectController.php

```
    /**
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project
     * @dontvalidate $project
     */
Public Function editAction ↲
  ( Tx_MittwaldTimetrack_Domain_Model_Project $project ) {
    $this->view->assign('project', $project);
    $this->view->assign('allProjects', $this->projectRepository->findAll());
}

    /**
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project
     */
```

```
Public Function updateAction ↲
  ( Tx_MittwaldTimetrack_Domain_Model_Project $project ) {
    $this->projectRepository->update($project);
    $this->redirect('show', NULL, NULL, Array('project'=>$project));
}
```

Wie wir sehen, entsprechen diese Methoden im weitesten der *new*- und der *create*-Aktion. Im Gegensatz zur *create*-Methode wird zum Speichern der Änderungen jedoch die update-Methode des Repositories aufgerufen.

Bei dem View für die *edit*-Aktion können wir uns großzügig an der *new*-Aktion bedienen:

PHP: Resources/Private/Templates/Project/edit.html

```
<h2>Projekt bearbeiten: {project.name}</h2>

<f:form action="update" object="{project}">
  <table>
    <tr>
      <td>Projektname:</td>
      <td><f:form.textbox property="name" /></td>
    </tr><tr>
      <td>Übergeordnetes Projekt:</td>
      <td>
        <f:form.select property="parent"
                       options="{allProjects}"
                       optionLabelField="name" />
      </td>
    </tr>
  </table>
  <f:form.submit value="Speichern" />
</f:form>
```

Natürlich ist diese Lösung nicht sonderlich elegant – schließlich haben wir das Bearbeitungsformular gänzlich unverändert einfach übernommen. Wir werden später in Abschnitt IV.2.2 auf Seite 120 Möglichkeiten betrachten, wie man solche Redundanzen in eigene Unter-Templates auslagern kann.

III.4.5.2 Validierung von Eingaben

III.4.5.2.a Benutzung von Validatoren

Bisher haben wir uns keine Gedanken darüber gemacht, wie und ob die Benutzereingaben eigentlich validiert werden. Nichts könnte einen Benutzer also daran hindern, einfach ein Projekt mit einem leeren Namen anzulegen, was in aller Regel nicht gewünscht sein dürfte.

Glücklicherweise stellt Extbase auch hierfür Methoden bereit, die uns jede Menge Arbeit abnehmen. Möchten wir ein einzelnes Attribut eines Domänenobjektes validieren, so können wir dies über eine einfache Anmerkung in dem DocComment-Kommentar des entsprechenden Attributes. Veranschaulichen wir dies anhand des Attributes *name* der *Projekt*-Klasse. Wir möchten, dass dieser mindestens drei Zeichen und höchstens 255 Zeichen lang sein darf:

PHP: Classes/Domain/Model/Project.php

```
    /**
     * @var string
     * @validate StringLength(minimum = 3, maximum = 255)
     */
Protected $name;
```

Wenn nun diesem Attribut ein neuer Wert zugewiesen wird (wie zum Beispiel über die von uns implementierte Methode *setName*), wird der neue Wert automatisch anhand dieser Regel validiert.

Wird nun ein nicht valides Objekt (also zum Beispiel ein Projekt ohne Namen) als Parameter an eine Controller-Aktion (wie zum Beispiel die *create*-Aktion) übergeben, so leitet uns der Controller automatisch wieder zu der Aktion zurück, von der wir gekommen sind (in diesem Fall zur *new*-Aktion).

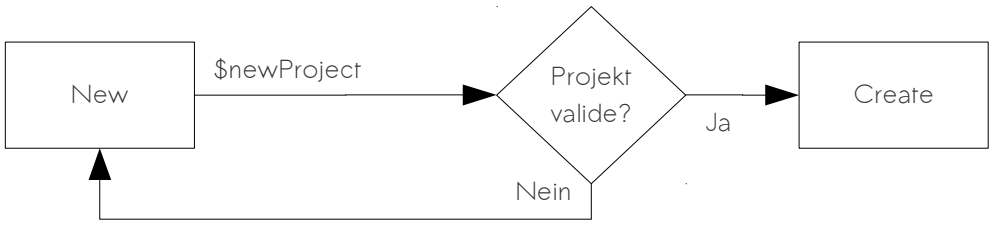

Abbildung 26: Ablauf bei der Validierung von Parametern

Der *StringLength*-Validator ist natürlich nicht der einzige Validator, der uns zur Verfügung steht. In jeder DocComment-Anmerkung können wir beliebig viele dieser Validatoren auf ein einzelnes Attribut anwenden:

```
    /**
     * @var string
     * @validate Alphanumeric, StringLength(minimum = 3, maximum = 64)
     */
Protected $fictionalProperty;
```

Insgesamt stehen folgende Validatoren zur Validierung von Objektattributen zur Verfügung:

Validator	Beschreibung	Beispiel
Alphanumeric	Überprüft, ob eine Zeichenkette nur aus Buchstaben (Groß- und Kleinschreibung) und Zahlen besteht.	`@validate Alphanumeric`
EmailAddress	Überprüft, ob eine Zeichenkette eine E-Mail-Adresse ist.	`@validate EmailAddress`
NotEmpty	Überprüft, ob ein Attribut entweder einer leere Zeichenkette oder NULL ist.	`@validate NotEmpty`
NumberRange	Ermittelt, ob ein Attribut einen numerischen Wert enthält und innerhalb eines bestimmten Intervalls befindet.	`@validate NumberRange (startRange = 10, endRange = 100)`
RegularExpression	Überprüft den Inhalt eines Attributs gegen einen regulären Ausdruck. Der Ausdruck muss vollständig mit Begrenzern angegeben werden.	`@validate RegularExpression (regularExpression = '/(Hello|Goodbye) World!/i')`
StringLength	Ermittelt, ob sich die Länge einer Zeichenkette innerhalb eines bestimmten Intervalls befindet.	`@validate StringLength (minimum=10, maximum=100)`

III.4.5.2.b Ausgabe von Fehlern

Eine gute Validierung der Eingabe ist trotz allem nur die halbe Miete. Falls die Eingaben eines Formulars die Validierung nicht bestehen, sollte dem Benutzer zum Beispiel angezeigt werden, was er eigentlich falsch gemacht hat. Andernfalls besteht die Gefahr, dass er das Formular einfach frustriert links liegen lässt.

Zu diesem Zweck weist Extbase allen Eingabefeldern, die einem nicht-validen Attribut zugeordnet sind, zunächst eine besondere CSS-Klasse zu (namentlich `f3-form-error`). Dies ermöglicht es, Eingabefelder mit fehlerhaften Eingaben, gesondert zu formatieren:

Abbildung 27: Fehlerbenachrichtung durch Formatierung des Eingabefelds

Sollte dies nicht ausreichen, stellt Fluid einen eigenen ViewHelper für das Darstellen von Fehlermeldungen zur Verfügung, den *Error*-ViewHelper. Diesen können wir verwenden wie folgt (platzierbar am besten innerhalb des Formulars, über den Eingabefeldern):

HTML: Resources/Private/Templates/Project/new.html

```
<!-- "for" muss dem Namen des Formulars entsprechen! //-->
<f:form.errors for="project">
  <div style="background-color: #fdd; padding: 12px;">
    <b>{error.propertyName}</b>
    <f:for each="{error.errors}" as="fieldError">
      <div>{fieldError.code}: {fieldError.message}</div>
    </f:for>
  </div>
</f:form.errors>
```

Dieses Konstrukt ist erforderlich, da die Fehlermeldungen in verschachtelter Form gespeichert werden. Zunächst werden sämtliche Fehlermeldungen durchgelaufen, die dem Formular mit dem Namen „project“ zugeordnet sind. Jedes dieser Objekte enthält wiederum alle Fehlermeldungen, die einem bestimmten Attribut zugeordnet sind. Diese werden wiederum separat mit einem *For*-ViewHelper durchlaufen. Letztendlich erhalten wir durch diese Konstruktion eine Ausgabe ähnlich der folgenden:

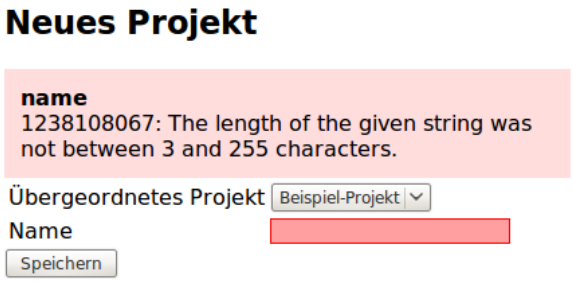

Abbildung 28: Fehlerbenachrichtung durch ausgelagerte Textmitteilung

Zur schöneren Gestaltung würde es sich schließlich anbieten, sowohl den Namen des betroffenen Attributes (bezeichnet durch `{error.propertyName}`) als auch die Fehlermeldungen selbst durch sprachabhängige Texte zu ersetzen:

```
<!-- "for" muss dem Namen des Formulars entsprechen! //-->
<f:form.errors for="project">
  <div style="background-color: #fdd; padding: 12px;">
    <b><f:translate key="Project_Attributes_{error.propertyName}" />:
</b>
    <f:for each="{error.errors}" as="fieldError">
      <f:translate
key="Project_Error_{error.propertyName}_{fieldError.code}" />
    </f:for>
  </div>
</f:form.errors>
```

Versuchen wir nun noch einmal, ein Projekt mit leerem Namen anzulegen:

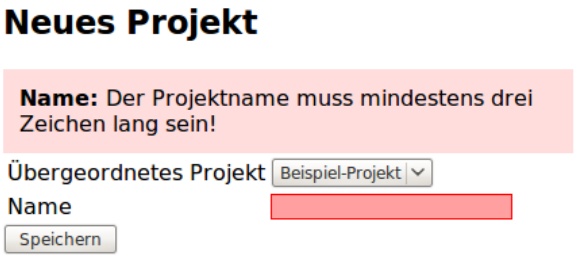

Abbildung 29: Fehlerbenachrichtung mit sprachabhängigen Textmeldungen

III.4.5.3 Rückmeldung geben: Flash Messages

So schön es auch ist, einem Benutzer mitteilen zu können, dass er etwas falsch gemacht hat; der Benutzer freut sich grundsätzlich mehr, wenn er eine Rückmeldung erhält, dass er etwas *richtig* gemacht hat. So wäre es doch zum Beispiel schön, wenn der Benutzer nach dem Anlegen eines neuen Projektes eine Meldung im Stil von „Herzlichen Glückwunsch, Ihr neues Projekt wurde erfolgreich erstellt“ lesen könnte. In Extbase können wir dies mit den sogenannten *Flash Messages* umsetzen. Dabei handelt es sich um kurze Rückmeldungen, die zum Beispiel im Bereich über der Projektübersicht angezeigt werden können.

Da die Nachricht erscheinen soll, nachdem ein Projekt erstellt wurde, müssen wir logischerweise in der *create*-Aktion Hand anlegen. Extbase bringt hierfür eine eigene Controller-Klasse mit sich, die *Tx_Extbase_MVC_Controller_FlashMessages*-Klasse. Eine Instanz dieser Klasse wird automatisch in jedem Controller in dem Attribut `flashMessages` gespeichert. Erweitern wir also die *create*-Methode wie folgt:

PHP: Classes/Controller/ProjectController.php

```
    /**
     * @param Tx_MittwaldTimetrack_Domain_Model_Project $project
     */
Public Function createAction (
  Tx_MittwaldTimetrack_Domain_Model_Project $project ) {
    $this->projectRepository->add($project);
    $this->flashMessage->add('Das Projekt '.$project->getName()
         . ' wurde erfolgreich angelegt.');
    $this->redirect('index');
}
```

Schließlich müssen wir nur noch festlegen, an welcher Stelle in der View die Flash Messages dann dargestellt werden sollen. Selbstverständlich stellt Fluid auch hierfür einen ViewHelper zur Verfügung:

HTML: Resources/Private/Templates/Project/index.html

```
<f:renderFlashMessages style="background-color: #dfd; padding: 12px;" />
```

Wenn nun ein neues Projekt angelegt wird (und dieses die Validierung erfolgreich besteht), so bekommt der Benutzer auf der Übersichtsseite anschließend eine entsprechende Bestätigungsnachricht präsentiert:

Abbildung 30: Rückmeldung über Flash Messages

III.4.5.4 Der letzte Schritt: Zeiten buchen

III.4.5.4.a Der Timeset-Controller

Kommen wir nun zum Kernelement der Zeiterfassungsanwendung: Dem Buchen von gearbeiteten Zeiten. Zu diesem Zweck werden wir im folgenden Abschnitt einen eigenen Controller für das Erstellen von Zeitpaaren implementieren. Zu diesem Zweck soll der momentan eingeloggte Benutzer ausgelesen werden und überprüft werden, ob dieser auch tatsächlich Mitglied in dem Projekt ist, für welches er eine Zeit buchen möchte.

Beginnen wir zunächst mit der Implementierung des *Timeset*-Controllers:

PHP: Classes/Controller/TimesetController.php

```
Class Tx_MittwaldTimetrack_Controller_TimesetController
    Extends Tx_Extbase_MVC_Controller_ActionController {

    Protected $userRepository;
    Protected $projectRepository;

    Public Function initializeAction() {
        $this->userRepository =& t3lib_div::makeInstance (
          'Tx_Extbase_Domain_Repository_FrontendUserRepository' );
        $this->projectRepository =& t3lib_div::makeInstance (
          'Tx_MittwaldTimetrack_Domain_Repository_ProjectRepository' );
    }

        /**
         * @param Tx_MittwaldTimetrack_Domain_Model_Project $project
         * @param Tx_MittwaldTimetrack_Domain_Model_Timeset $timeset
         * @dontvalidate $timeset
         */
    Public Function newAction (
        Tx_MittwaldTimetrack_Domain_Model_Project $project
```

```
        Tx_MittwaldTimetrack_Domain_Model_Timeset $timeset = NULL ) {

        $user = $this->userRepository->findByUid (
          $GLOBALS['TSFE']->fe_user->user['uid'] );
        $assignment = $user ? $project->getAssignmentForUser($user) : NULL;

        If($assignment === NULL) Return '';

        $this->view->assign('project', $project)
                   ->assign('timeset', $timeset)
                   ->assign('user', $user);

    }

        /**
         * @param Tx_MittwaldTimetrack_Domain_Model_Project $project
         * @param Tx_MittwaldTimetrack_Domain_Model_Timeset $timeset
         */
    Public Function createAction (
        Tx_MittwaldTimetrack_Domain_Model_Project $project
        Tx_MittwaldTimetrack_Domain_Model_Timeset $timeset ) {

        $user = $this->userRepository->findByUid (
          $GLOBALS['TSFE']->fe_user->user['uid'] );
        $assignment = $user ? $project->getAssignmentForUser($user) : NULL;

        If($assignment === NULL) Return '';

        $assignment->addTimeset($timeset);
        $assignment->getProject->addAssignment($assignment);
        $this->projectRepository->update ( $timeset->getProject() );

        $this->flashMessages->add (
          'Die Zeitbuchung wurde erfolgreich durchgeführt' );
        $this->redirect (
          'show', 'Project', NULL, Array('project' => $timeset->getProject());
    }
}
```

Zunächst zur Erläuterung der *new*-Aktion: Wir beginnen damit, den derzeit eingeloggten Benutzer anhand des globalen TSFE-Objektes auszulesen. Glücklicherweise stellt Extbase bereits fertige Klassen für die Abbildung von Frontendbenutzern sowie ein Repository dafür bereit. Dieses Objekt dient dazu, zu ermitteln, ob der Benutzer überhaupt ein Mitglied des ausgewählten Projektes ist.

Stellen wir nun fest, dass der Benutzer gar kein Mitglied des Projektes ist, so brechen wir an dieser Stelle mit einem einfachen `Return '';` ab. Sicherlich gibt es elegantere Methoden dafür, doch für den Anfang soll diese Methode erst einmal ausreichen. Später werden wir elegantere Methoden zur Fehlerbehandlung betrachten. Schließlich weisen wir das neue Zeitpaar und das ausgewählte Projekt dem View zu – den wir auch noch erstellen müssen.

In der *create*-Aktion führen wir zunächst dieselben Sicherheitsüberprüfungen wie in der *new*-Aktion durch – sicher ist sicher. Anschließend fügen wir der Projektmitgliedschaft das neue Zeitpaar hinzu und aktualisieren das zugehörige Projekt über das Projekt-Repository (wir erinnern uns: Da es sich bei dem Projekt um eine Aggregatwurzel handelt, werden beim Update alle assoziierten Objekte – also auch das neue Zeitpaar – zusammen mit dem Projekt aktualisiert).

III.4.5.4.b Views für die Formulare

Zu guter Letzt können wir nun die View für die *new*-Aktion erstellen. Da diese nicht zum *ProjectController*, sondern nun zum *TimesetController* gehört, wird diese nun im Verzeichnis *Resources/Private/Templates/Timeset/* gespeichert:

HTML: Resources/Private/Templates/Timeset/new.html

```
<h2>Neue Zeitbuchung: {project.name}</h2>

<f:form action="create" object="{timeset}">
  <table>
    <tr>
      <td>Benutzer:</td>
      <td>{user.name}</td>
    </tr><tr>
      <td>Projekt:</td>
      <td>{project.name}</td>
    </tr><tr>
      <td>Startzeit:</td>
      <td><f:form.textbox property="starttime" /></td>
    </tr><tr>
      <td>Stopzeit:</td>
      <td><f:form.textbox property="stoptime" /></td>
    </tr><tr>
      <td>Kommentar:</td>
      <td><f:form.textarea property="comment" rows="5" cols="80" /></td>
    </tr>
  </table>
  <f:form.submit value="Zeit buchen" />
  <f:form.hidden name="project" value="{project}" />
</f:form>
```

Damit man das Formular für neue Zeitpaare auch erreichen kann, fügen wir noch einen Link in die *show*-Aktion des Projekt-Controllers ein:

HTML: Resources/Private/Templates/Project/show.html

```
<ul>
  <li>
    <f:link.action controller="Timeset" action="new" arguments="{project :
project}">
      Neue Zeitbuchung
    </f:link.action>
  </li>
</ul>
```

III.4.5.4.c Validierung eines Zeitpaares

Zum Abschluss können wir unser Zeitpaar-Modell so derart anpassen, dass die Benutzereingaben zumindest halbwegs validiert werden. So zum Beispiel, dass die Angabe einer Start- und einer Stoppzeit sowie eines Kommentars zwingend erforderlich ist:

PHP: Classes/Domain/Model/Timeset.php

```
    /**
     * The assignment
     * @var Tx_MittwaldTimetrack_Domain_Model_Assignment
     */
Protected $assignment;

    /**
     * The start time of this timeset
     * @var DateTime
     * @validate NotEmpty
     */
Protected $starttime;

    /**
     * The stop time of this timeset
     * @var DateTime
     * @validate NotEmpty
     */
Protected $stoptime;

    /**
     * A comment for this timeset
     * @var string
     * @validate StringLength(minimum=3)
     */
Protected $comment;
```

Selbstverständlich gibt es noch viele weitere Aspekte, die an solch einem Zeitpaar eigentlich überprüft werden müssten, bevor es tatsächlich gespeichert werden darf. Was zum Beispiel passiert, wenn die Stoppzeit *vor* der Startzeit liegt? Mit all diesen Möglichkeiten zur erweiterten Validierung beschäftigen wir uns in Abschnitt IV.5.1 auf Seite 143.

Betrachten wir nun zuletzt das soeben gebaute Formular. Bei den Start- und Stopp-Feldern können wir nun eine Zeit in jedem Format eintragen, das von der PHP-Funktion *strtotime*[14] verstanden wird.

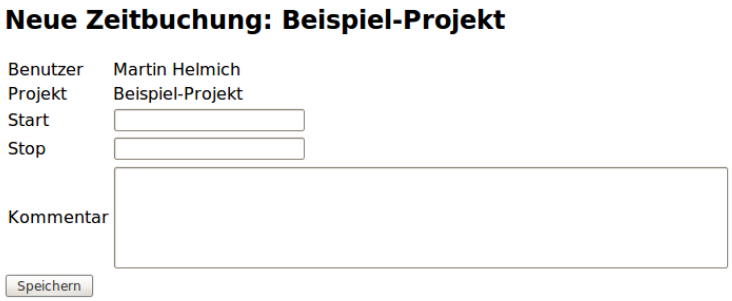

Abbildung 31: Das Formular zum Eintragen eines neuen Zeitpaars

14 Siehe http://de.php.net/manual/de/function.strtotime.php

III.5 Zusammenfassung

Fassen wir die Arbeitsschritte des letzten Kapitels noch einmal kurz zusammen:

1. Wir beginnen mit der grundlegenden Konfiguration der Erweiterung. Dies beinhaltet die Erstellung der Datenbankstruktur in der Datei *ext_tables.sql* und die Konfiguration der Tabellen in den Dateien *ext_tables.php* und *Configuration/TCA/[Project/Assignment/…].php*.
2. Zu diesem Zeitpunkt können Sie die Erweiterung bereits über den Erweiterungsmanager installieren. Die benötigten Tabellen werden automatisch in der Datenbank angelegt. Über das TYPO3-Backend können Sie nun schon einige Daten zum Testen anlegen. Anschließend können Sie das Plugin auf einer beliebigen Seite im TYPO3-Seitenbaum einfügen.
3. Im nächsten Schritt implementieren wir die Anwendungsdomäne. Diese besteht aus dem Modell im Verzeichnis *Classes/Domain/Model/* und den Repositories im Verzeichnis *Classes/Domain/Repository/*. Jede Datenbankspalte wird als Klassenattribut abgebildet; für auslesbare bzw. bearbeitbare Attribute müssen sondierende bzw. manipulierende Methoden implementiert werden. Des weiteren können wir über Anmerkungen in den Kommentaren einfache Validierungsregeln festlegen.
4. Nachdem die Domäne implementiert wurde, können wir die benötigten Controller im Verzeichnis *Classes/Controller/* erstellen. Für jede Controller-Aktion wird eine eigene Methode in der Controller-Klasse implementiert. Außerdem wird für jede Aktion ein eigener View als HTML-Datei im Verzeichnis *Resources/Private/Templates/[Controllername]/* angelegt.

Teil IV: Vertiefung

Zu diesem Zeitpunkt wissen Sie bereits alles, um Ihre erste Erweiterung komplett mit Extbase umzusetzen. Betrachten wir das letzte Kapitel noch einmal im Rückblick: Nach der Einrichtung der allgemeinen Konfigurationsdateien (*ext_localconf.php*, *ext_tables.php*, und so weiter) begannen wir zunächst mit der Umsetzung des Domänenmodells – da wir uns bereits zuvor Gedanken über das Design der Anwendung gemacht hatten, blieb hier nur noch die bloße Umsetzung – und anschließend mit der Erstellung der Controller und der dazugehörigen Views. Außerdem haben wir einen Blick auf die Methoden geworfen, die Extbase zur Validierung von Benutzereingaben anbietet.

All diese Kenntnisse reichen bereits für viele Erweiterungen aus. Dennoch wird Ihnen aufgefallen sein, dass im Verlauf des letzten Kapitels immer wieder Fragen offen geblieben sind. Wie zum Beispiel können wir es verhindern, dass der Benutzer beim Buchen eines Zeitpaares eine Stoppzeit angibt, die vor der Startzeit liegt? Können wir die Projekte in der Übersichtsansicht alphabetisch sortieren? Gibt es eine elegante Möglichkeit, die Arbeitszeiten angemessen zu formatieren? Wie können wir es vermeiden, ein und dasselbe Formular zum Neuanlegen und Bearbeiten von Projekten doppelt zu speichern? Um all diese Fragen zu klären, müssen wir einen tieferen Blick in die Mechanismen von Extbase und Fluid werfen.

Im folgenden Kapitel werden wir zunächst einige fortgeschrittene Methoden zur Darstellung von Daten betrachten. Anschließend möchten wir herausfinden, wie wir eigentlich selbst Objekte aus der Datenbank laden können und kompliziertere Validierungen durchführen können. Anschließend werden wir uns der Erstellung eines Backend-Moduls widmen – in den Zeiten vor Extbase jedes Mal eine nervtötende Angelegenheit – und schließlich noch einen kurzen Blick hinter die Fassaden von Extbase werfen.

IV.1 Manuelles Erstellen der Extension

Mit dem Extbase-Kickstarter lassen sich viele mühsame Arbeitsschritte einsparen. Dennoch wird es hin und wieder notwendig sein, selbst Hand an die Dateien anzulegen, die vom Kickstarter erstellt wurden.

Um ein tiefer greifendes Verständnis über den Aufbau einer Extbase-Erweiterung zu vermitteln, werden wir im folgenden Abschnitt die Zeiterfassungs-Erweiterung aus dem ersten Kapitel noch einmal auf „dem Fußweg“ anlegen.

IV.1.1 Grundlegende Dateien

Beginnen wir also damit, das Verzeichnis *mittwald_timetrack/* unterhalb des *typo3conf/ext/*-Verzeichnisses anzulegen. Im nächsten Schritt werden wir die grundlegenden Dateien anlegen, die vom TYPO3-Kern in jeder Erweiterung verlangt werden, namentlich die *ext_emconf.php*, die *ext_localconf.php* sowie die *ext_tables.php*.

Die *ext_emconf.php* enthält Informationen über die jeweilige Erweiterung. Diese werden vom Erweiterungsmanager im TYPO3-Backend ausgelesen. Die Angaben über Abhängigkeiten und Konflikte veranlassen den Erweiterungsmanager gegebenenfalls, vorher noch weitere Erweiterungen zu installieren oder zu deinstallieren.

Füllen wir nun die *ext_emconf.php* mit unseren Beispieldaten:

PHP: ext_emconf.php

```
<?php
$EM_CONF[$_EXTKEY] = Array (
    'title'              => 'Timetracking',
    'description'        => 'A timetracking extension to demonstrate Extbase',
    'category'           => 'example',
    'shy'                => 0,
    'version'            => '1.0.0',
    'dependencies'       => '',
    'conflicts'          => '',
    'priority'           => '',
    'loadOrder'          => '',
    'module'             => '',
    'state'              => 'stable',
    'uploadfolder'       => 0,
    'createDirs'         => '',
    'modify_tables'      => '',
    'clearcacheonload'   => 1,
    'lockType'           => '',
    'author'             => 'Ihr Name',
```

```
    'author_email'       => 'ihre.email@domain.tld',
    'author_company'     => 'Ihre Firma',
    'CGLcompliance'      => '',
    'CGLcompliance_note' => '',
    'constraints' => Array( 'depends' => Array( 'php'     => '5.2.0-0.0.0',
                                                'typo3'   => '4.3.dev-4.4.99',
                                                'extbase' => '1.0.1-0.0.0',
                                                'fluid'   => '1.0.1-0.0.0' ),
                            'conflicts' => Array ( ),
                            'suggests'  => Array ( ) ),
    '_md5_values_when_last_written' => '',
);
?>
```

Wichtig ist vor allem das Array-Element mit dem Index „constraints“. Diese Angaben teilen dem Extensionmanager mit, dass er bei der Installation dieser Erweiterung auch automatisch Extbase und Fluid installieren muss.

Die *ext_localconf.php* enthält grundlegende Konfigurationseinstellungen für unsere Erweiterung, in der *ext_tables.php* stehen weiterführende Konfigurationen. Im Moment reicht es, wenn wir die *ext_localconf.php* mit dem leeren Standardinhalt anlegen:

PHP: ext_localconf.php

```
<?php
If(!defined('TYPO3_MODE')) Die ('Access denied.');
?>
```

In der *ext_tables.php* können wir bereits eine Zeile einfügen, die das Verzeichnis *Configuration/Typoscript* als statisches Template anmeldet:

PHP: ext_tables.php

```
<?php
If(!defined('TYPO3_MODE')) Die ('Access denied.');
t3lib_extMgm::addStaticFile (
    $_EXTKEY, 'Configuration/TypoScript', 'Time tracking' );
?>
```

IV.1.2 Erstellen eines Plugins

Genau wie pi_base-Erweiterungen kann eine Extbase-Erweiterung sogenannte *Plugins* zur Verfügung stellen. Dies sind besondere Inhaltselemente, die als Seiteninhalt auf TYPO3-Seiten eingefügt werden können. Im folgenden Abschnitt werden wir nun so ein Plugin einrichten, damit wir unsere Erweiterung auf einer Seite im Seitenbaum einfügen können.

Die Einrichtung eines Plugins besteht aus zwei Schritten: Zunächst muss TYPO3 mitgeteilt werden, dass das Plugin überhaupt existiert. Anschließend muss das Plugin konfiguriert werden. Zur Konfiguration gehört zum Beispiel, auf welche Controller das Plugin zugreifen darf, und welcher Controller standardmäßig aufgerufen wird.

> **Hintergrund:** Bei pi_base-Erweiterungen entsprach ein Plugin stets einer Klasse. In unserem Fall hätte der Kickstarter also einen Ordner *pi1/* angelegt, in welchem eine Klasse mit dem Namen *tx_mittwaldtimetrack_pi1* liegen würde. Dies ist mit Extbase-Erweiterungen nicht mehr der Fall. Hier definiert sich ein Plugin einfach als eine Gruppe von Controllern, deren Views auf einer bestimmten Seite angezeigt werden dürfen. Eine gesonderte Klasse für jedes Plugin ist nicht mehr notwendig. Mehr über die technischen Hintergründe findet sich im Abschnitt IV.7 über den Extbase-Dispatcher auf Seite 153.

IV.1.2.1 Registrierung des Plugins

Extbase stellt sowohl für die Registrierung als auch für die Konfiguration eines Plugins die notwendigen Methoden bereit. Die Registrierung eines Plugins erfolgt mit der statischen Methode *registerPlugin* der Klasse *Tx_Extbase_Utility_Extension*:

PHP: EXT:extbase/Classes/Utility/Extension.php

```
public static function registerPlugin (
    $extensionName, $pluginName, $pluginTitle );
```

Mit dieser Methode können wir unser Plugin in der *ext_tables.php* registrieren. Fügen Sie hierzu den folgenden Code unten in die bereits bestehende Datei *ext_tables.php* ein:

PHP: ext_tables.php

```
Tx_Extbase_Utility_Extension::registerPlugin (
    $_EXTKEY, 'Pi1', 'A timetracking extension' );
```

IV.1.2.2 Konfiguration des Plugins

Auch für die Konfiguration von Plugins stellt die Klasse *Tx_Extbase_Utility_-Extension* eine Methode bereit:

PHP: EXT:extbase/Classes/Utility/Extension.php

```
public static function configurePlugin (
    $extensionName,
    $pluginName,
    array $controllerActions,
    array $nonCachableControllerActions = array() );
```

Diese Methode erwartet als Parameter zunächst den Namen der Extension und des Plugins, sowie anschließend eine Liste von Controllern und Aktionen, auf die das Plugin zugreifen darf. Da wir uns in Kapitel II.2.3.3 auf Seite 38 bereits ausführlich Gedanken darüber gemacht haben, welche Controller wir benötigen, ist dies nun nicht weiter kompliziert.

> Hinweis: Die Angabe der zur Verfügung stehenden Controller ist essentiell für das Rechtemanagement. Wie wir später feststellen werden, können dieselben Controller auch zur Steuerung eines Backendmoduls verwendet werden. Nun wäre es natürlich fatal, wenn ein nicht angemeldeter Besucher im Frontend einfach einen der Backendcontroller aufrufen könnte. Aus diesem Grund macht es durchaus Sinn, dass die zugänglichen Controller explizit angegeben werden müssen.

Der letzte Parameter schließlich steuert das Caching-Verhalten unseres Plugins. Während bei pi_base-Plugins das Caching nur für das ganze Plugin aktiviert oder deaktiviert werden konnte (über USER und USER_INT), kann bei einem Extbase-Plugin das Caching für einzelne Controller-Aktionen gezielt deaktiviert werden. So macht es zum Beispiel Sinn, Übersichts- oder Detailansichten zu cachen, während Formulare oder Darstellungen mit sich häufig aktualisierenden Inhalten nicht gecacht werden sollten.

Die beiden Parameter zur Steuerung des Controllerverhaltens sind Arrays, die jeweils den Controllernamen als Index verwenden und die betroffenen Aktionen als kommaseparierte Liste. Wir können unser Plugin also mit der folgenden Anweisung in der *ext_localconf.php* konfigurieren:

PHP: ext_localconf.php

```
Tx_Extbase_Utility_Extension::configurePlugin (
    $_EXTKEY,
    'Pi1',
    Array ( 'Project' => 'index,show,new,create,delete,edit,update',
            'Timeset' => 'new' ),
    Array ( 'Project' => 'index,show,new,create,delete,edit,update',
            'Timeset' => 'new' )
);
```

Auf die Bedeutung der einzelnen Controller werden wir später noch genauer eingehen. Gleichzeitig erscheint es sinnvoll, dass unsere Erweiterung zu keinem Zeitpunkt gecacht wird, da sich die mit einer Zeiterfassungsanwendung verwalteten Daten häufig ändern. Aus diesem Grund wurden sämtliche sichtbaren Controlleraktionen auch als nicht-gecacht gekennzeichnet.

IV.1.3 Konfigurieren der Datenbanktabellen

IV.1.3.1 Datenbankstruktur

Über die Datenbankstruktur unserer Erweiterung hatten wir uns bereits in Abschnitt II.2.4 auf Seite 39 Gedanken gemacht. Jedoch sollten wir unsere Tabellen nach den Konventionen von Extbase benennen, wenn wir uns späteren Konfigurationsaufwand ersparen möchten (wir erinnern uns: *Convention over Configuration*!).

Unsere Fachmodell-Klassen werden später per Konvention den Namen *Tx_Mittwald-Timetrack_Domain_Model_Project* usw. tragen. Nun gibt es eine weitere Konvention, die besagt, dass die Datenbanktabellen für ein Fachmodell dieselben Namen tragen, wie die zugehörige Klasse, nur in Lowercase. Daher besteht unsere Datenbankstruktur aus den folgenden Klassen:

- *tx_mittwaldtimetrack_domain_model_project*
- *tx_mittwaldtimetrack_domain_model_role*
- *tx_mittwaldtimetrack_domain_model_assignment*
- *tx_mittwaldtimetrack_domain_model_timeset*

In der Datei *ext_tables.sql* geben wir nun für jede dieser Tabellen jeweils einen SQL-Dump zum Erstellen der Tabelle an:

SQL: ext_tables.sql

```
CREATE TABLE tx_mittwaldtimetrack_domain_model_project (
    uid INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
    pid INT(11) DEFAULT '0' NOT NULL,

    tstamp INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    crdate INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    deleted TINYINT(4) UNSIGNED DEFAULT '0' NOT NULL,
    hidden TINYINT(4) UNSIGNED DEFAULT '0' NOT NULL,

    name VARCHAR(255) DEFAULT '' NOT NULL,
    project INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    start INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    end INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    assignments INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    children INT(11) UNSIGNED DEFAULT '0' NOT NULL,

    PRIMARY KEY (uid),
    KEY pid (pid),
    KEY parent (parent)
);
```

```
CREATE TABLE tx_mittwaldtimetrack_domain_model_role (
    uid INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
    pid INT(11) DEFAULT 0 NOT NULL,

    tstamp INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    crdate INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    deleted TINYINT(4) UNSIGNED DEFAULT '0' NOT NULL,
    hidden TINYINT(4) UNSIGNED DEFAULT '0' NOT NULL,

    name VARCHAR(128) DEFAULT '' NOT NULL,

    PRIMARY KEY (uid),
    KEY pid (pid)
);

CREATE TABLE tx_mittwaldtimetrack_domain_model_timeset (
    uid INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
    pid INT(11) DEFAULT 0 NOT NULL,

    tstamp INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    crdate INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    deleted TINYINT(4) UNSIGNED DEFAULT '0' NOT NULL,
    hidden TINYINT(4) UNSIGNED DEFAULT '0' NOT NULL,

    assignment INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    starttime INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    stoptime INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    comment TEXT DEFAULT '' NOT NULL,

    PRIMARY KEY (uid),
    KEY pid (pid),
    KEY assignment (assignment_uid)
);

CREATE TABLE tx_mittwaldtimetrack_domain_model_assignment (
    uid INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,
    pid INT(11) DEFAULT 0 NOT NULL,

    tstamp INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    crdate INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    deleted TINYINT(4) UNSIGNED DEFAULT '0' NOT NULL,
    hidden TINYINT(4) UNSIGNED DEFAULT '0' NOT NULL,

    user INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    project INT(11) UNSIGNED DEFAULT '0' NOT NULL,
    role TINYINT(1) UNSIGNED DEFAULT '0' NOT NULL,
    timesets INT(11) UNSIGNED DEFAULT '0' NOT NULL,

    PRIMARY KEY (uid),
```

```
    KEY pid (pid),
    KEY user (user),
    KEY project (project)
);
```

IV.1.3.2 Metadaten

Im nächsten Schritt müssen wir TYPO3 von der Existenz der neuen Tabellen in Kenntnis setzen. Dies geschieht durch eine entsprechende Konfiguration in dem sogenannten *Table Configuration Array*, kurz TCA. Mit der Einführung von Extbase wird das TCA nicht mehr nur für die Darstellung von Formularen im TYPO3-Backend, sondern auch für das objektrelationale Mapping auf die Domänenobjekte verwendet (wir erinnern uns an den Abschnitt III.1.4.1 über das Data-Mapping auf Seite 49). Die korrekte Konfiguration dieses Arrays ist also unerlässlich geworden.

Zunächst konfigurieren wir einige allgemeine Metadaten für die jeweiligen Tabellen in der *ext_tables.php*. Anschließend folgen die einzelnen Spaltendefinitionen in der TCA-Dateien (dazu mehr im nächsten Kapitel).

> **Hinweis:** Die Konfigurationseinstellungen in der *ext_tables.php* und *tca.php* sind keine Extbase-Neuheiten und werden als zumindest im Groben bekannt vorausgesetzt. Für weiterführende Dokumentation über das *Table Configuration Array* sei an dieser Stelle auf die offizielle Dokumentation in der TYPO3-API hingewiesen. Diese findet sich unter:
>
> http://typo3.org/documentation/document-library/core-documentation/doc_core_api/4.2.0/view/4/1/

Fügen wir also folgenden Code in die *ext_tables.php* ein:

PHP: ext_tables.php

```
$lPath =
'LLL:EXT:mittwald_timetrack/Resources/Private/Language/locallang_db.xml';
$ePath = t3lib_extMgm::extPath($_EXTKEY);
$TCA['tx_mittwaldtimetrack_domain_model_project'] = Array (
  'ctrl' => array (
    'title'             => $lPath .
':tx_mittwaldtimetrack_domain_model_project',
    'label'             => 'name',
    'tstamp'            => 'tstamp',
    'crdate'            => 'crdate',
    'delete'            => 'deleted',
    'enablecolumns'     => Array ( 'disabled' => 'hidden' ),
    'dynamicConfigFile' => $ePath . 'Configuration/TCA/Project.php',
    'iconfile'          => $ePath . 'Resources/Public/Icons/'
                           . 'tx_mittwaldtimetrack_domain_model_project.gif' )
```

```
);

$TCA['tx_mittwaldtimetrack_domain_model_role'] = Array (
  'ctrl' => array (
    'title'             => $lPath . ':tx_mittwaldtimetrack_domain_model_role',
    'label'             => 'user',
    'tstamp'            => 'tstamp',
    'crdate'            => 'crdate',
    'delete'            => 'deleted',
    'enablecolumns'     => Array( 'disabled' => 'hidden' ),
    'dynamicConfigFile' => $ePath . 'Configuration/TCA/Role.php',
    'iconfile'          => $ePath .
'Resources/Public/Icons/tx_mittwaldtimetrack_role.png'
) );

$TCA['tx_mittwaldtimetrack_domain_model_assignment'] = Array (
  'ctrl' => array (
    'title'             => $lPath .
':tx_mittwaldtimetrack_domain_model_assignment',
    'label'             => 'user',
    'tstamp'            => 'tstamp',
    'crdate'            => 'crdate',
    'delete'            => 'deleted',
    'enablecolumns'     => Array( 'disabled' => 'hidden' ),
    'dynamicConfigFile' => $ePath . 'Configuration/TCA/Assignment.php',
    'iconfile'          => $ePath . 'Resources/Public/Icons/'
                           .
'tx_mittwaldtimetrack_domain_model_assignment.gif' ) );

$TCA['tx_mittwaldtimetrack_domain_model_timeset'] = Array (
  'ctrl' => Array (
    'title'             => $lPath .
':tx_mittwaldtimetrack_domain_model_timeset',
    'label'             => 'assignment',
    'tstamp'            => 'tstamp',
    'crdate'            => 'crdate',
    'delete'            => 'deleted',
    'enablecolumns'     => Array ( 'disabled' => 'hidden' ),
    'dynamicConfigFile' => $ePath . 'Configuration/TCA/Timeset.php',
    'iconfile'          => $ePath . 'Resources/Public/Icons/'
                           . 'tx_mittwaldtimetrack_domain_model_timeset.gif' )
);
```

IV.1.3.3 Spaltenkonfigurationen

In der Konfiguration in *ext_tables.php* sind bereits mehrfach Bezüge auf diverse Dateien im Verzeichnis *Configuration/TCA/* enthalten. Dieses enthält Definitionen für die einzelnen Attribute der Fachobjekte. Seit Extbase werden diese Definitionen auch von Extbase für das objektrelationale Mapping herangezogen. Dies bedeutet, dass Extbase das TCA auswertet, um festzustellen, welche Datenbankspalte welchem Objektattribut zuzuordnen ist. Auch Assoziationen zwischen Objekten werden aus dem TCA ermittelt. Aus diesem Grund ist es wichtig, das TCA sorgfältig zu konfigurieren, damit wir später keine Probleme bekommen.

Zugegebenermaßen ist es mühselig, diese Konfiguration auf dem „Fußweg“ zu erarbeiten. Allein schon, um Fehler zu vermeiden, sollte aus diesem Grund sofern möglich stets der Kickstarter verwendet werden. Da das gesamte TCA, das sich hier über mehrere Dateien verteilt, einige hundert Zeilen lang ist und nur wenige für uns interessante Punkte enthält, sehen wir an dieser Stelle davon ab, diese in voller Länge darzustellen. Auch das Anlegen der entsprechenden *locallang_db.xml* sollte nicht vergessen werden. Sämtliche Dateien finden sich in voller Länge in dem Extension-Paket, welches zusammen mit diesem Dokument herunter geladen werden kann.[15]

Besondere Aufmerksamkeit sollte man der Definition von Assoziationen zwischen Klassen widmen. Extbase wertet hierfür alle Spalten aus, bei denen im `config`-Bereich das Attribut `foreign_table` gesetzt ist (wir erinnern uns hierfür an den Abschnitt III.1.4.1 über das Data-Mapping auf Seite 49). Für das `project`-Attribut des *Project*-Objektes sähe die Spaltendefinitionen im TCA beispielsweise so aus:

PHP: Configuration/TCA/tca.php

```
'project' => Array(
    'exclude' => 1,
    'label'   =>
'LLL:EXT:mittwald_timetrack/Resources/Private/Language/locallang_db.xml'
                  . ':tx_mittwaldtimetrack_domain_model_project.project',
    'config'  => Array (
        'type'  => 'select',
        'items' => Array(Array('LLL:EXT:mittwald_timetrack/Resources/Private/'
                               . 'Language/locallang_db.xml:'
                               . 'tx_mittwaldtimetrack_domain_model_project.'
                               . 'project.null',
                              ''),
                         Array('', '--div--') ),
        'foreign_table' => 'tx_mittwaldtimetrack_domain_model_project',
```

15 Siehe http://www.mittwald.de/extbase-dokumentation/

```
            'size' => 1,
            'maxitems' => 1 )
    ),
```

Tatsächlich ist der Spaltentyp, welcher mit der Eigenschaft *config.type* beschrieben wird, für Extbase sogar unerheblich. Solange wir nicht den Wert *passthrough* wählen, interessiert Extbase lediglich, ob die Eigenschaft *config.foreign_table* gesetzt wurde. Außerdem verwendet Extbase das Attribut *config.maxitems*, um zu ermitteln, wie viele Objekte überhaupt zu dieser Assoziation gehören. Wenn sich die Assoziation auf mehr als ein Objekt bezieht, so wird diese später auf ein ganzes Array mit Objekten abgebildet, ansonsten auf ein einzelnes Objekt.

IV.1.3.4 Zusammenfassung

Da die Konfiguration des Datenmodells – gerade ohne Kickstarter – nicht gerade trivial ist, und eine gewisse Menge an Arbeitsschritten erfordert, fassen wir die notwendigen Schritte an dieser Stelle noch einmal zusammen:

1. Die Struktur der Datenbank wird als SQL-Dump in der Datei *ext_tables.sql* gespeichert.
2. In der Datei *ext_tables.php* werden die einzelnen Tabellen konfiguriert. Dies ist notwendig, damit wir die Objekte über das TYPO3-Backend bearbeiten können und Extbase die Datenbanktabellen auf Klassen abbilden kann.
3. Schließlich werden in der Datei *Configuration/TCA/tca.php* die Spalten der einzelnen Tabellen konfiguriert. Diese Konfiguration ermöglicht Extbase das Abbilden der Datenbankspalten auf einzelne Klassenattribute. Der Name dieser Konfigurationsdatei ist frei wählbar; er muss lediglich in der *ext_tables.php* korrekt angegeben werden.
4. Die sprachabhängigen Bezeichnungen für die Datenbanktabellen werden in der Datei *Resources/Private/Language/locallang_db.xml* gespeichert. Dieser Dateiname ist frei wählbar und kann in der *ext_tables.php* und *tca.php* angegeben werden.
5. Icons für die Tabellen können im Verzeichnis *Resources/Public/Icons* gespeichert werden.

IV.2 Views für Fortgeschrittene

IV.2.1 Layouts

Häufig liegt mehreren verschiedenen Views innerhalb einer Erweiterung eine ähnliche Struktur zugrunde. Nun ist es natürlich unhandlich und unflexibel, diese Struktur (zum Beispiel eine Menüleiste, die oberhalb des eigentlichen Inhaltes platziert ist, oder eine Fußzeile mit Copyright-Informationen) in jedem View wieder einzeln nachzubauen: Möchte man diese Struktur nachträglich einmal ändern – dies kommt spätestens dann zwangsläufig vor, wenn ein Administrator eine Erweiterung in seine eigene Seite integrieren möchte – dann muss diese in jedem View einzeln angepasst werden.

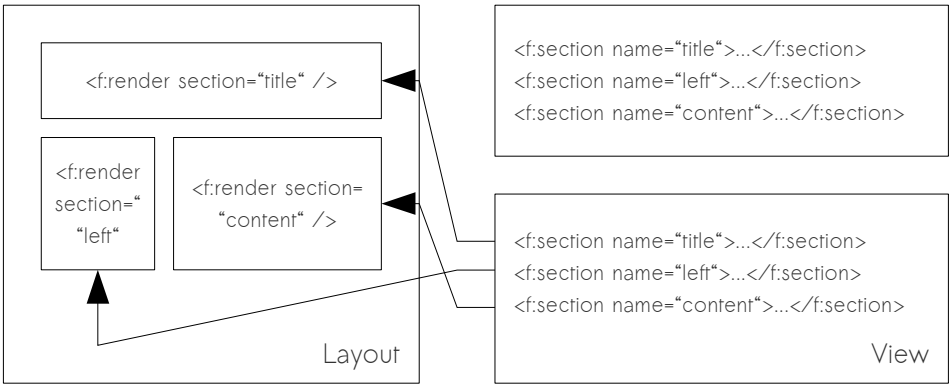

Abbildung 32: Bearbeitung von Layouts

Um dies zu vermeiden, bietet Fluid sogenannte *Layouts* an. In einem Layout können allgemeine Strukturmerkmale ausgelagert und so von mehreren Views gleichzeitig genutzt werden. So hatten wir in der Zeiterfassungserweiterung aus dem letzten Kapitel häufig eine Titelzeile, darunter ein Optionsmenü und schließlich den eigentlichen Inhalt. Nun könnte man hieraus ein Layout bauen, welches aussieht wie folgt:

HTML: Resources/Private/Layouts/default.html

```
<h2><f:render section="title" /></h2>

<div id="mittwaldtimetrack-menu">
  <f:render section="menu" />
</div>

<f:renderFlashMessages style="background-color: #dfd; padding: 12px;" />
```

```
<f:render section="content" />

<div id="mittwaldtimetrack-footer">
  Powered by MittwaldTimetrack
</div>
```

Layouts werden in dem Verzeichnis *Resources/Private/Layouts* gespeichert. Ein Layout ist im Gegensatz zu einem View nicht an einen bestimmten Controller gebunden, sondern kann von Views aller Controller benutzt werden. Des weiteren können wir in einem Layout alle ViewHelper benutzen, die wir auch in einem regulären View verwenden können.

In einem View kann ein Layout mithilfe entsprechender ViewHelper eingebunden werden. Nehmen wir die Projektübersicht aus der Zeiterfassungserweiterung als Beispiel:

HTML: Resources/Private/Templates/Project/index.html

```
<f:layout name="default" />
<f:section name="title">Projekte</f:section>

<f:section name="menu">
  <ul>
    <li><f:link.action action="new">Neues Projekt</f:link.action></li>
  </ul>
</f:section>

<f:section name="content">
  <table>
    <tr>
      <th>Projektname</th>
      <th>Anzahl Mitarbeiter</th>
      <th>Aufgewendete Zeit</th>
    </tr>
    <f:if condition="{projects}">
      <f:then>
        <f:for each="{projects}" as="project">
          <tr>
            <td>{project.name}</td>
            <td><f:count subject="{project.assignments}" /> Mitarbeiter</td>
            <td>{project.workedTime}<td>
          </tr>
        </f:for>
      </f:then><f:else>
        <tr><td>Keine Projekte gefunden</td></tr>
      </f:else>
    </f:if>
  </table>
</f:section>
```

Mit der Anweisung

```
<f:layout name="default" />
```

können wir also spezifizieren, welches Layout für diese View verwendet werden soll. Anschließend können wir mithilfe des *Section*-ViewHelpers angeben, mit welchen Inhalten die Abschnitte im Layout gefüllt werden sollen:

```
<f:section name="content">
  <!-- ... //-->
</f:section>
```

IV.2.2 Partielle Templates

Zuweilen kommt es nicht nur vor, dass wir bestimmte Layouts in mehreren Views wiederverwenden wollen, sondern auch bestimmte inhaltliche Elemente, wie zum Beispiel Formulare oder ähnliche Komponenten. Bei der Umsetzung der Formulare zum Erstellen und zum Bearbeiten von Projekten unserer Zeiterfassungserweiterung war bereits aufgefallen, dass genau dasselbe Formular in zwei verschiedenen Views benutzt wurde. Auch diese Redundanz ist genau genommen überflüssig.

Genau wie bei den Layouts bietet Fluid auch hier eine Möglichkeit an, solche häufig verwendeten Teilvorlagen (Partielle Vorlagen oder engl. *partial templates*) auszulagern. Diese werden in dem Verzeichnis *Resources/Private/Partials* gespeichert. Eine Teilvorlage für das eben erwähnte Projektformular könnte also beispielsweise so aussehen:

HTML: Resources/Private/Partials/projectForm.html

```
<f:form action="{action}" object="{project}">

  <f:form.errors for="project">
    <div style="background-color: #fdd; padding: 12px;">
      <b><f:translate key="Project_Attributes_{error.propertyName}" />: </b>
      <f:for each="{error.errors}" as="fieldError">
        <f:translate
key="Project_Error_{error.propertyName}_{fieldError.code}" />
      </f:for>
    </div>
  </f:form.errors>

  <table>
    <tr>
      <td>Projektname:</td>
      <td><f:form.textbox property="name" /></td>
    </tr><tr>
      <td>Übergeordnetes Projekt:</td>
```

```
        <td>
          <f:form.select property="parent"
                         options="{allProjects}"
                         optionLabelField="name" />
        </td>
      </tr>
    </table>
    <f:form.submit value="Speichern" />
  </f:form>
```

Tatsächlich wurde dieses Formular eins zu eins aus dem *new*-View des Projekt-Controllers übernommen. Nun können wir allerdings sowohl den *new*- als auch den *edit*-View so anpassen, dass für das Formular das soeben erstelle Teil-Template verwendet wird. Hierzu verwenden wir den *Render*-ViewHelper. Diesem teilen wir mit dem Attribut *partial* den Namen des zu verwendenden Teil-Templates mit. Des weiteren können wir über das Attribut *arguments* zusätzliche Parameter übergeben. Dies ist auch notwendig, da die Variablen, die dem View zugewiesen wurden, nicht automatisch an die Teilvorlage weitervererbt werden. Daher müssen wir dem Partial alle Variablen explizit neu zuweisen.

HTML: Resources/Private/Templates/Project/new.html

```
<h2>Neues Projekt</h2>

<f:render partial="projectForm"
          arguments="{action : 'create', project : project,
allProjects : allProjects}" />
```

HTML: Resources/Private/Templates/Project/edit.html

```
<h2>Projekt bearbeiten: {project.name}</h2>

<f:render partial="projectForm"
          arguments="{action : 'update', project : project,
allProjects : allProjects}" />
```

Sollte es uns nun irgendwann einmal in den Sinn kommen, das Formular grundlegend zu überarbeiten, so muss nun nur noch an einer einzigen Stelle Hand angelegt werden.

IV.2.3 Eigene View-Helper

Mit Layouts und Partials stehen uns bereits mächtige Werkzeuge zur Verfügung, um sich oft wiederholende Komponenten aus Vorlagen auszulagern. Dennoch sind auch die technischen Möglichkeiten dieser Komponenten irgendwann einmal erschöpft. In diesem Fall können wir uns mit einem eigenen View-Helper aushelfen.

IV.2.3.1 Einstieg

Ein Kernkonzept von Fluid besteht darin, dass die komplette Ausgabelogik innerhalb von View-Helpern angesiedelt ist. Dies betrifft zum Beispiel die Kontrollstrukturen wie *If/Then/Else* und *For*, mit denen wir bereits in den vorigen Abschnitten gearbeitet haben. Dabei wird jeder View-Helper mit einer eigenen PHP-Klasse abgebildet. Die einzige Aufgabe von Fluid ist es, Aufrufe solcher View-Helper im Template zu erkennen und den richtigen View-Helper aufzurufen.

Da jeder View-Helper also eine eigene Klasse ist, können wir problemlos eigene View-Helper hinzufügen. Diese sind innerhalb einer Erweiterung unter dem Verzeichnis *Classes/ViewHelpers* gespeichert.

Bei der Umsetzung der Zeiterfassungsanwendung im vorigen Kapitel hatten wir mit diversen Schönheitsfehlern zu kämpfen. So wurde die Arbeitszeit im Datenmodell zum Beispiel in Sekunden errechnet, und folglich auch in Sekunden im Frontend ausgegeben. Nun wäre es natürlich schön, wenn die Ausgabe im Frontend in Stunden oder Tagen erfolgen könnte. Noch besser wäre es natürlich, wenn die Formatierung von der tatsächlichen Menge der gearbeiteten Zeit abhängen würde – wenn zum Beispiel alle Zeiten bis 48 Stunden als Stunden, und alles darüber als Tage angezeigt würde.

Fluid stellt für eigene View-Helper eine abstrakte Basisklasse bereit, namentlich die Klasse *Tx_Fluid_Core_ViewHelper_AbstractViewHelper*. Unser eigener View-Helper – nennen wir ihn *TimeFormatViewHelper* – muss also auf dieser Klasse aufbauen:

PHP: Classes/ViewHelpers/TimeFormatViewHelper.php

```
Class Tx_MittwaldTimetrack_ViewHelpers_TimeFormatViewHelper
    Extends Tx_Fluid_Core_ViewHelpers_AbstractViewHelper {

        /**
         * @param int $amount
         * @return string
         */
    Public Function render($amount) {
        If($amount >= 172800)
            Return number_format($amount / 86400.00, 2, ',', '').' Tage';
        Else
            Return number_format($amount / 3600.00, 2, ',', '').' Stunden';
    }

}
```

Jeder View-Helper muss eine Methode mit dem Namen render zur Verfügung stellen. Diese muss eine Zeichenkette zurückliefern, welche dann im View ausgegeben wird. Sämtliche Parameter, die diese Methode entgegennimmt und für die kein Standardwert definiert ist, müssen (!) dem View-Helper im Template als Parameter übergeben werden. Des weiteren sind auch hier die Quelltextkommentare von großer Wichtigkeit.

Damit sich die View-Helper unterschiedlicher Erweiterungen nicht in die Quere kommen, sind diese innerhalb sogenannter Namensräume organisiert. Sämtliche View-Helper, die von Fluid selbst bereitgestellt werden, befinden sich im Namensraum *f* (aus diesem Grund beginnt auch jeder Aufruf eines View-Helpers mit `f:`). Um unseren eigenen View-Helper verwenden zu können, muss innerhalb des Views zunächst ein Namensraum dafür deklariert werden:

```
{namespace mwt=Tx_MittwaldTimetrack_ViewHelpers}
```

Unpraktischerweise ist es derzeit nicht möglich, eine Namensraumdeklaration in ein Layout auszulagern. Diese Deklaration muss in jedem View und sogar in jedem Partial erneut erfolgen. Nachdem der Namensraum deklariert wurde, kann nun mit der gewohnten Syntax auf alle View-Helper darin zugegriffen werden:

```
<mwt:timeFormat amount="12345" />
  <!-- Ausgabe: 3,42 Stunden //-->

<mwt:timeFormat amount="1234567" />
  <!-- Ausgabe: 14,28 Tage //-->
```

Fügen wir diesen View-Helper nun beispielsweise in die index-View der Zeiterfassungserweiterung ein, so sieht das Ganze schon gleich viel ansprechender aus:

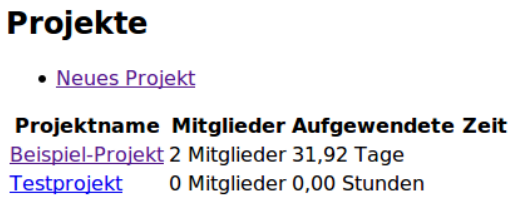

Abbildung 33: Der eigene View-Helper im Einsatz

IV.2.3.2 Eigene Formularfelder

Mit diesem einfachen View-Helper ist die Menge der Möglichkeiten natürlich längst noch nicht erschöpft. So können wir zum Beispiel beim Erstellen eines View-Helpers auch vollen Gebrauch der Vererbung machen, und so zum Beispiel einen bereits existierenden View-Helper verbessern oder unseren Bedürfnissen anpassen.

Wenn wir zum Beispiel die Zeiterfassungserweiterung noch einmal betrachten, so könnte auffallen, dass wir bisher noch keine Möglichkeit haben, einem Projekt Benutzer zuzuordnen. Dies möchten wir folgendermaßen umsetzen: Im Bearbeitungsformular für ein neues oder bereits existierendes Projekt soll eine Liste aller Benutzer angezeigt werden – wenn viele Mitarbeiter beteiligt sind, zweifellos nicht die beste Lösung; an dieser Stelle soll es jedoch ausreichen. Jedem diesem Benutzer kann dann in einem Auswahlfeld eine – oder auch keine – Rolle zugewiesen.

Zu diesem Zweck erweitern wir den Select-ViewHelper, der von Fluid zur Verfügung gestellt wird:

PHP: Classes/ViewHelpers/Form/UserRoleViewHelper.php

```
Class Tx_MittwaldTimetrack_ViewHelpers_Form_UserRoleViewHelper
  Extends Tx_Fluid_ViewHelpers_Form_SelectViewHelper {

    Public Function initializeArguments() {
        parent::initializeArguments();
        $this->registerArgument (
          'project', 'Tx_MittwaldTimetrack_Domain_Model_Project', '', TRUE);
        $this->registerArgument (
          'user', 'Tx_Extbase_Domain_Model_FrontendUser', '', TRUE);
    }

    Protected Function getOptions() {
        $options = Array(0 => 'Kein Mitglied');
        ForEach($this->arguments['options'] As $option) {
            If($option InstanceOf Tx_MittwaldTimetrack_Domain_Model_Role)
                $options[$option->getUid()] = $option->getName();
        } Return $options;
    }

    Protected Function getSelectedValue() {
        $assignment = $this->arguments['project']
          ? $this->arguments['project']->getAssignmentForUser($this-
>arguments['user'])
          : NULL;
        Return $assignment ? $assignment->getRole()->getUid() : 0;
    }
```

```
    Protected Function getName() {
        Return parent::getName().'['.$this->arguments['user']->getUid().']';
    }

}
```

Sämtliche Methoden aus dieser Klasse überschreiben Methoden aus der Elternklasse *Tx_Fluid_ViewHelpers_Form_SelectViewHelper*. Da wir die *render*-Methode nicht überschreiben möchten, können wir zusätzliche Argumente für den View-Helper auch in der *initializeArguments*-Methode definieren. Diese ist definiert wie folgt:

```
protected function registerArgument (
  $name, $type, $description, $required = FALSE, $defaultValue = NULL );
```

Der neue View-Helper nimmt als Argumente also zusätzlich ein Projekt und einen einzelnen Benutzer entgegen.

Die *getOptions*-Methode wurde so modifiziert, dass zusätzlich zu den angegebenen Optionen noch eine leere Option mit der Beschriftung „Kein Mitglied“ dargestellt wird. Außerdem werden nur Optionen dargestellt, die eine Instanz des Rollen-Domänenobjektes sind.

Die *getSelectedValue*-Methode schließlich ermittelt die Rolle, die der angegebene Benutzer in dem angegebenen Projekt derzeit einnimmt, und liefert anschließend die UID dieser Benutzerrolle zurück.

Damit wir mit diesem View-Helper arbeiten können, müssen wir den entsprechenden Views noch jeweils eine Liste mit allen Benutzern sowie allen Benutzerrollen zur Verfügung stellen. Dies betrifft die *new*- und den *edit*-View des Projekt-Controllers:

```
$this->view->assign( 'users', $this->userRepository->findAll() )
           ->assign( 'roles', $this->roleRepository->findAll() );
```

In der View schließlich können wir alle Benutzer in einer Schleife durchlaufen und für jeden dieser Benutzer unseren neuen View-Helper verwenden:

HTML: Resources/Private/Partials/projectForm.html

```
<tr>
  <td>Mitarbeiter</td>
  <td>
    <table>
       <f:for each="{users}" as="user">
        <tr>
          <td>{user.name}</td>
          <td>
              <mwt:form.userRole name = "assignments"  project =
"{project}"
```

```
                                    user = "{user}"       options =
"{roles}"   />
          </td>
        </tr>
      </f:for>
    </table>
  </td>
</tr>
```

Hier ist zu beachten, dass dieser View-Helper nicht einem bestimmten Attribut des Projektes zugeordnet ist. Wir müssen nun also auch noch den Projekt-Controller anpassen, sodass dieser zusätzliche Parameter nun ausgewertet wird:

PHP: Classes/Controller/ProjectController.php

```
Public Function createAction ( Tx_MittwaldTimetrack_Domain_Model_Project
$project,
                              Array
$assignments ) {

    $project->removeAllAssignments();
     ForEach($assignments As $userId => $roleId) {
         If($roleId == 0) Continue;
         $project->assignUser($this->userRepository->findByUid((int)
$userId),
                               $this->roleRepository->findByUid((int)
$roleId) );
     }

    $this->projectRepository->add($project);
    $this->flashMessages->add('Das Projekt '.$project->getName()
      . ' wurde erfolgreich angelegt.');
    $this->redirect('index');

}
```

Die update-Aktion passen wir ebenfalls dementsprechend an. Betrachten wir nun das Formular zum Anlegen eines neuen Projektes im Frontend:

Abbildung 34: Das eigene Formularelement in Aktion

IV.2.4 Zugriff auf TYPO3-Content-Elemente

In einer pi_base-Erweiterung war es recht einfach, auf die Typoscript-API zurückzugreifen, um komplexe Objekte für die Ausgabe im Frontend zu erstellen. Da diese über Typoscript konfigurierbar waren, konnten sie problemlos von Administratoren an die eigenen Bedürfnisse angepasst werden, ohne den eigentlichen Quelltext der Erweiterung modifizieren zu müssen. Hierzu noch einmal ein kleines Beispiel: Hatten wir im Typoscript-Setup einer Erweiterung beispielsweise folgenden Code,

```
plugin.tx_myext.object = IMAGE
plugin.tx_myext.object {
  file = EXT:myext/res/images/logo.png
  file.maxH = 100
  altText = Mein Logo
}
```

so konnte der Entwickler innerhalb der Erweiterung mit folgendem Codeabschnitt auf dieses Typoscript-Objekt zugreifen und es ausgeben:

```
$content = $this->cObj->cObjGetSingle ( $this->conf['object'],
                                        $this->conf['object.] );
```

Diese Möglichkeit steht in Fluid nicht mehr ohne weiteres zur Verfügung, da in dem Template selbst kein PHP-Code untergebracht werden kann. Nun könnte man selbstverständlich innerhalb des Controllers eine Instanz der *tslib_cObj*-Klasse erstellen und einfach den erstellten Inhalt an den View weitergeben. Nur wäre dies zum einen eine grobe Verletzung des MVC-Musters (die Erstellung der Ausgabe hat im Controller schlicht nichts zu suchen), des weiteren würde eine Ausgabe, die HTML-Code beinhaltet, an der Validierung von Fluid scheitern.

Um dieses Problem zu lösen, bietet Fluid auch hierfür einen View-Helper an – was auch sonst? Der CObject-ViewHelper nimmt einen Typoscript-Pfad sowie optional ein Array mit Daten als Parameter entgegen, und rendert daraus mithilfe der Typoscript-API das entsprechende Objekt. Um den oben angegebenen Typoscript-Code also ausgeben zu können, verwenden wir folgenden View-Helper:

```
<f:cObject typoscriptObjectPath="plugin.tx_myext.object" />
```

IV.2.5 Eigene Views

Bisher haben wir die Ausgabe von Inhalten mit Fluid als die einzig mögliche betrachtet. Tatsächlich ist die *TemplateView*-Klasse, die von Fluid bereitgestellt wird, nur eine von vielen möglichen Schnittstellen.

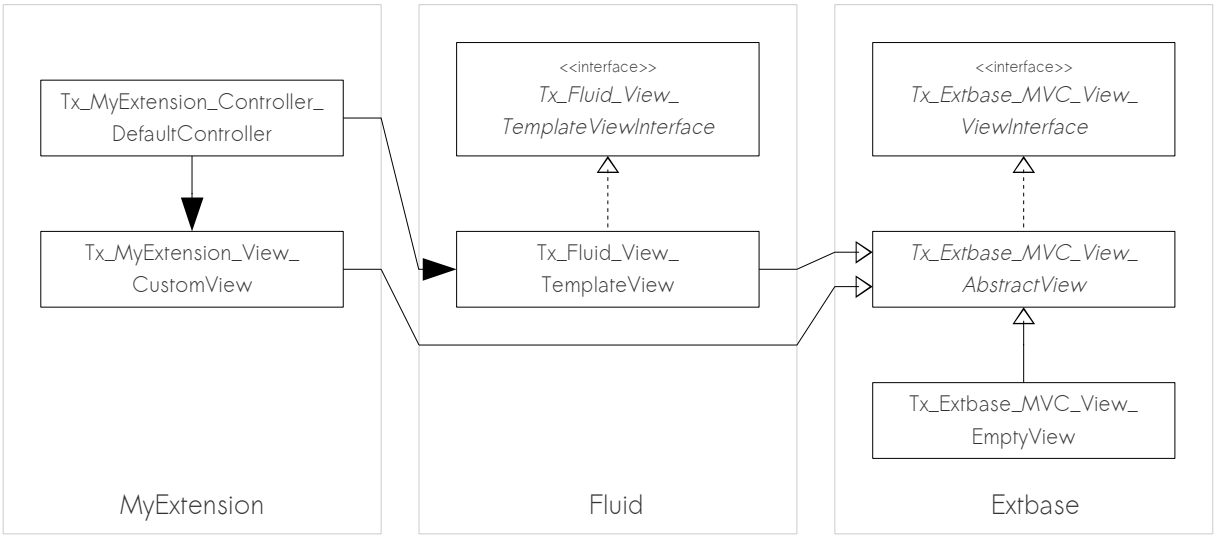

Abbildung 35: Vererbungsschema der View-Klassen

Wird in einem *ActionController* eine Aktion aufgerufen, so sucht dieser zunächst nach einem Template für diese Aktion im *Resources*-Verzeichnis. Wird dieses Template nicht gefunden, wird nach einer View-Klasse gesucht, die nach dem Schema

Tx_[Extensionname]_View_[Controllername]_[Aktion]

benannt ist. Dieses Schema kann mit dem Controller-Attribut `$viewObjectNamePattern` überschrieben werden. Des weiteren kann mit dem Attribut `$defaultViewObjectName` eine Standard-View-Klasse definiert werden. Sollte dies immer noch nicht ausreichen, kann im eigenen Controller auch die Methode *resolveView* komplett überschrieben werden.

Jede eigene View-Klasse sollte die Schnittstelle *Tx_Extbase_MVC_View_View-Interface* implementieren. Unterklassen der Klasse *Tx_Extbase_MVC_View_Abstract-View* tun dies automatisch. Demnach muss jeder View eine Methode *render* zur Verfügung stellen. Diese muss eine Zeichenkette zurück liefern, welche dann schließlich im Frontend ausgegeben werden kann.

Eine solche Klasse könnte zum Beispiel so aussehen:

```
Class Tx_MyExtension_View_Default_IndexView Extends
Tx_Extbase_MVC_View_AbstractView {
    Public Function render() {
        $content  = print_r($this->viewData, TRUE);
        $content .= "My very first own view!";
        Return $content;
    }
}
```

Dieser View wird nun standardmäßig verwendet, wenn für die *index*-Aktion des *DefaultControllers* kein Template definiert ist. Um diese View-Klasse für alle Aktionen zu verwenden, kann innerhalb des Controllers das Attribut `$defaultViewObjectName` überschrieben werden:

```
Class Tx_MyExtension_Controller_DefaultController
  Extends Tx_Extbase_MVC_Controller_ActionController {

    Protected $defaultViewObjectName =
'Tx_MyExtension_View_Default_IndexView';

    /* Insert action methods here... */

}
```

Einen eigenen View zu implementieren lohnt sich beispielsweise dann, wenn die Anforderungen an die Ausgabe einer Anwendung so komplex oder speziell sind, dass Fluid tatsächlich an seine Grenzen stößt (was zugegebenermaßen nicht so schnell passieren dürfte), oder wenn der Entwickler eine andere Template-Engine als Fluid verwenden möchte.

IV.3 Zugriff auf die Datenbank mit Repositories

Bei dem Zugriff auf die Datenbank haben wir uns bisher auf die *findBy**-Methoden der Repository-Objekte beschränkt. In den allermeisten Fällen reichen diese jedoch nicht für alle Anforderungen aus. So stoßen wir bereits an Probleme, wenn wir Objekte in einer bestimmten Reihenfolge laden, oder assoziierte Objekte mit berücksichtigen wollen. Diese Probleme zu lösen, soll Ziel des folgenden Abschnitts sein.

Erinnern wir uns noch einmal an die Grundsätze des Domain-Driven Design. Demnach erfolgt der einzige Zugriff auf die Infrastruktur – also die Datenbank – über Repository-Objekte. Innerhalb des Domänenmodells oder der Anwendungsschicht haben Zugriffe auf die Datenbank nichts verloren.

IV.3.1 Einführung in das Query Object Model

FLOW3 und Extbase bringen ein eigenes Abstraktionsmodell für den Zugriff auf eine Datenbank mit sich, das sogenannte *Query Object Model* – kurz QOM genannt. In diesem Modell wird eine einzelne Datenbankabfrage als Objekt abgebildet, welches verschiedene Methoden zum Durchführen von Datenbankoperationen zur Verfügung stellt.

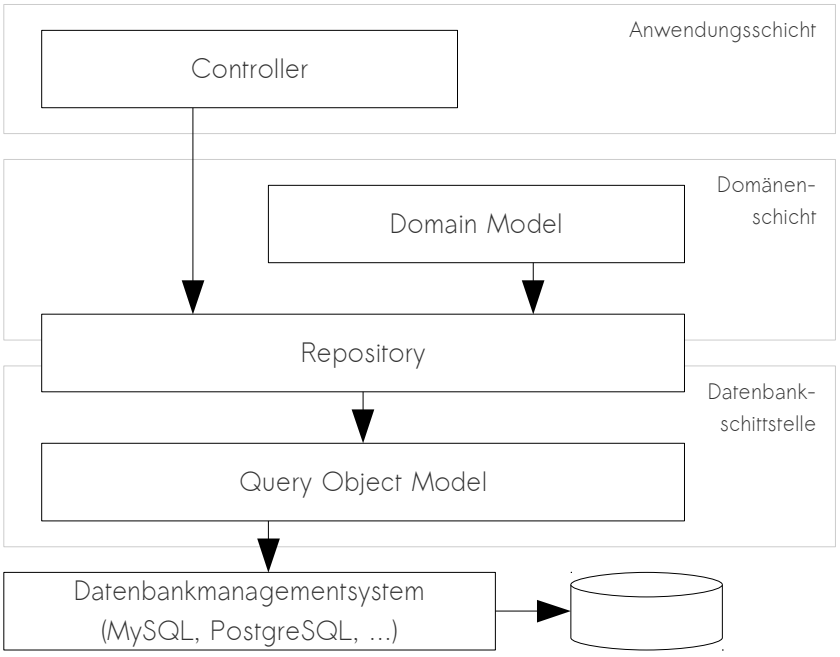

Abbildung 36: Das Query Object Model innerhalb der Extbase-Architektur

Innerhalb eines Repositories können wir ein solches *Query*-Objekt mit der Methode *createQuery* erstellen. Die so erstellte Abfrage ist bereits so konfiguriert, dass die zu dem Repository gehörende Datenbanktabelle abgefragt wird. Für die nachfolgenden Beispiele werden wir in der Zeiterfassungserweiterung aus dem vorherigen Kapitel verbleiben.

Der einfachste Fall ist eine Datenbankabfrage ohne jede Restriktion. Um solch eine Abfrage durchzuführen, reicht es aus, eine neue Datenbankabfrage zu erstellen und diese sofort auszuführen:

PHP: Classes/Domain/Repository/ProjectRepository.php

```
Class Tx_MittwaldTimetrack_Domain_Repository_ProjectRepository
  Extends Tx_Extbase_Persistence_Repository {

    Public Function findAll() {
        Return $this->createQuery()->execute();
    }

}
```

Tatsächlich macht auch die *findAll*-Methode, welche bereits von der *Tx_Extbase_Persistence_Repository*-Klasse zur Verfügung gestellt wird, nichts anderes. Erwähnt werden sollte noch, dass diese Methode ein Array zurück liefert, welches bereits die fertigen Domänenobjekte enthält. Die Zuordnung der Objektattribute (das sogenannte *data mapping*) wird bereits automatisch von Extbase durchgeführt.

IV.3.2 Datenbankoperationen im QOM

Nachdem solch eine einfache Abfrage nur in den seltensten Fällen ausreicht, werden wir im folgenden Abschnitt einige der elementaren Datenbankoperationen betrachten, und wie diese im *Query Object Model* abgebildet werden.

Grundkenntnisse über die relationale Algebra und die Funktionsweise von relationalen Datenbankmanagementsystemen werden hier als bekannt vorausgesetzt.

IV.3.2.1 Restriktionen

Als eine *Restriktion* wird eine Einschränkung der Ergebnismenge einer Datenbankabfrage bezeichnet. Es werden also nur diejenigen Datenbanktupel in die Ergebnismenge aufgenommen, die eine bestimmte Bedingung erfüllen. Diese Bedingungen können beliebig durch logische Operatoren („und“, „oder“ und „nicht“) verknüpft werden.

IV.3.2.1.a Einfache Restriktionen

Eine Restriktion (engl. *constraint*) wird im *Query Object Model* als eigenes Objekt abgebildet. Diese kann der Abfrage dann über die *matching*-Methode zugewiesen werden. Sämtliche Restriktionen können über entsprechende Methoden der *Query*-Klasse erstellt werden. Dabei muss immer eine Spalte angegeben werden, auf die die Restriktion angewandt werden soll, sowie ein Operand, also ein Vergleichswert.

Wollen also wir zum Beispiel nur Projekte mit einem bestimmten Namen aus der Datenbank laden, so können wir zunächst mit

```
$query = $this->createQuery();
$constraint = $query->equals('name', 'Projekt 1');
```

eine Restriktion erstellen. Diese kann anschließend mit

```
$query->matching($constraint);
```

der Datenbankabfrage zugewiesen werden. Um das ganze kürzer abbilden zu können, lässt sich dies natürlich auch verschachteln beziehungsweise verketten:

```
$query = $this->createQuery();
Return $query->matching($query->equals('name','Projekt 1'))
  ->execute();

    # Entspricht:
    #   SELECT * FROM tx_mittwaldtimetrack_domain_model_project
    #     WHERE name='Projekt 1';
```

Eine kleine Besonderheit: Bei der *equals*-Restriktion kann als Vergleichswert auch ein Array übergeben werden. In diesem Fall ergibt die Restriktion ein wahres Ergebnis, wenn ein Attribut mit einem dieser Werte übereinstimmt:

```
return $query->matching($query->equals('name',array('A','B')))
  ->execute();

    # Entspricht:
    #   SELECT * FROM tx_mittwaldtimetrack_domain_model_project
    #     WHERE name IN ('A','B');
```

Die Abfrageklasse stellt die Vergleichsmethoden *equals*, *like*, *lessThan*, *lessThanOrEqual*, *greaterThan* und *greaterThanOrEqual* zur Verfügung. Die Bedeutung der jeweiligen Methoden sollte sich von selbst erklären.

IV.3.2.1.b Logische Negation von Restriktionen

Auf jede Restriktion können *logische Operatoren* (sogenannte *Junktoren*) angewandt werden. Einer dieser Operatoren ist die Negation, ein einstelliger Operator. Mit der Negation wird der Wahrheitswert einer einzelnen Aussage umkehrt. Dies bedeutet, dass aus einer an sich wahren Restriktion eine unwahre wird, und aus einer falschen Restriktion eine wahre. Hierzu stellt die *Query*-Klasse die Methode *logicalNot* zur Verfügung. Hatten wir im vorigen Abschnitt alle Projekte mit einem bestimmten Namen geladen, so können wir diese Restriktion nun negieren, sodass wir alle Projekte enthalten, deren Name nicht mit dem Operand übereinstimmt:

```
Return $query->matching(
  $query->logicalNot($query->equals('name', 'Project 1'))
)->execute();

    # Entspricht:
    #   SELECT * FROM tx_mittwaldtimetrack_domain_model_project
    #     WHERE name != 'Project 1'
```

IV.3.2.1.c Logische Verknüpfung mehrerer Restriktionen

Weitere logische Operatoren sind die *Konjunktion* und die *Disjunktion*, auch als *logisches Und* und *logisches Oder* bezeichnet. Die *Query*-Klasse stellt hierfür die Methoden *logicalAnd* und *logicalOr* bereit, die beide jeweils zwei einzelne Restriktionen als Parameter erwarten. Möchten wir zum Beispiel alle Projekte laden, die kein übergeordnetes Projekt haben, oder deren Name „Projekt 1“ ist, so können wir folgende Abfrage verwenden:

```
Return $query->matching(
  $query->logicalOr( $query->equals('parent', 0),
                     $query->equals('name', 'Project 1') )
)->execute();

    # Entspricht:
    #   SELECT * FROM tx_mittwaldtimetrack_domain_model_project
    #     WHERE parent = 0 OR name = 'Project 1';
```

Solche Restriktionen lassen sich natürlich beliebig tief verschachteln, hier anhand einer fiktiven Tabelle mit den Spalten a bis d:

```
Return $query->matching(
  $query->logicalAnd(
    $query->logicalAnd(
      $query->logicalOr( $query->lessThanOrEqual('a', 5),
                         $query->greaterThan('b', 10) ),
```

```
      $query->equals('c', 0)
    ),
    $query->logicalNot($query->like('d', '%Hallo%'))
  )
);

    # Entspricht in SQL:
    #   SELECT * FROM fictional_table
    #     WHERE (a <= 5 OR b > 10) AND c = 0 AND d NOT LIKE "%Hallo%"
```

IV.3.2.2 Sortierung

Um die Sortierung der abgefragten Daten festzulegen, stellt das *Query*-Objekt die Methode *setOrderings* zur Verfügung. Diese akzeptiert als Parameter ein Array, in welchem die Spaltennamen als Indices und die Sortierreihenfolgen (auf- und absteigend) als Werte verwendet werden. Für die Sortierreihenfolge werden die Konstanten

1. `Tx_Extbase_Persistence_Query::ORDER_ASCENDING` und
2. `Tx_Extbase_Persistence_Query::ORDER_DESCENDING`

zur Verfügung gestellt. Wollen wir also zum Beispiel alle Projekte sortiert nach dem Projektnamen aus der Datenbank laden, so können wir dies mit folgendem Befehl erreichen:

```
Return $query->setOrderings (
  Array('name' => Tx_Extbase_Persistence_Query::ORDER_ASCENDING)
)->execute();

    # Entspricht in SQL:
    #   SELECT * FROM tx_mittwaldtimetrack_domain_model_project
    #     ORDER BY name ASC
```

Selbstverständlich kann eine solche Sortierung in beliebiger Kombination mit Restriktionen verwendet werden:

```
Return $query->matching (
    $query->logicalNot( $query->equals('name','Projekt 1') )
  ->setOrderings (
    Array('name' => Tx_Extbase_Persistence_Query::ORDER_ASCENDING) )
  ->execute();
```

IV.3.2.3 Ergebnismenge begrenzen

Häufig liefern Datenbankabfragen eine große Menge an Ergebnissen zurück. In diesem Fall werden wir manchmal vielleicht nur einen Teil dieser Menge verwenden wollen. Mit den Methoden *setLimit* und *setOffset* können wir die Ergebnismenge, die die Datenbankabfrage zurückliefert, einschränken. Dies ist zum Beispiel nützlich, wenn wir eine Seitennavigation implementieren möchten.

Dabei legt *setLimit* die Anzahl der Datensätze fest, die geladen werden sollen, während wir mit *setOffset* bestimmen, ab dem wievielten Datensatz die Objekte überhaupt geladen werden sollen. Folgender Befehl liefert zum Beispiel zehn Projekte ab dem zwanzigsten Projekt zurück:

```
Return $query->setLimit(10)
  ->setOffset(20)
  ->execute();
```

IV.3.2.4 Joins

Bei einem *Join* werden mehrere Tabellen (in der relationalen Algebra auch als Relation bezeichnet) miteinander kombiniert. Da FLOW3 – im Gegensatz zu Extbase – auf einem *Content Repository* nach der JCR-Spezifikation, und nicht auf einer relationalen Datenbank aufsetzt, ist die Möglichkeit, komplexe Joins zu umzusetzen, nur sehr halbherzig implementiert. Tatsächlich hat die derzeitige Extbase-Version (Stand 16. 2. 2010, TYPO3 4.3.1) in dieser Hinsicht noch keine produktionsfertige Schnittstelle.

Grundsätzlich gilt jedoch folgendes: Das *Query*-Objekt arbeitet mit einer gewissen Datenquelle, welche als eine Instanz der *Tx_Extbase_Persistence_QOM_SourceInterface*-Schnittstelle abgebildet wird. Dies kann mit der Klasse *Tx_Extbase_Persistence_QOM_Selector* entweder eine einzelne Tabelle, oder mit der Klasse *Tx_Extbase_Persistence_QOM_Join* ein Join aus zwei verschiedenen Tabellen sein. Um nun im *Query Object Model* einen Join abbilden zu können, müssen wir zunächst die Standard-Datenquelle der Abfrage mit einem Join ersetzen. Nachfolgend ein Versuch, nur Projekte zu laden, bei denen ein bestimmter Benutzer Mitglied ist:

```
$qomFactory = $this->persistenceManager->getBackend()->getQOMFactory();
$join = New Tx_Extbase_Persistence_QOM_Join(
  $qomFactory->selector ( 'Tx_MittwaldTimetrack_Domain_Model_Project',
                          'tx_mittwaldtimetrack_domain_model_project' ),
  $qomFactory->selector ( 'Tx_MittwaldTimetrack_Domain_Model_Assignment',
                          'tx_mittwaldtimetrack_domain_model_assignment' ),
  Tx_Extbase_Persistence_QOM_QueryObjectModelConstantsInterface
    ::JCR_JOIN_TYPE_INNER,
```

```
    $qomFactory->equiJoinCondition (
      'tx_mittwaldtimetrack_domain_model_project', 'uid',
      'tx_mittwaldtimetrack_domain_model_assignment', 'project' )
);

$query->setSource($join);
$query->matching($query->equals('user', $user));
$query->execute();
```

Dieses zugegebenermaßen abenteuerliche Konstrukt erstellt zwar einen korrekten Join, dennoch ist dieser leider nicht zu gebrauchen, da wir keine Spaltenprojektion vornehmen können, wir also die auszuwählenden Spalten nicht einschränken können. Da hierdurch einige Spaltennamen doppelt verwendet werden (wie zum Beispiel die `uid`-Spalte), kann das *Data Mapping* nicht mehr fehlerfrei durchgeführt werden. Diese Methode eignet sich somit höchstens dann, wenn wir die Ergebnisdatensätze nur zählen, aber nicht mit den einzelnen Objekten weiterarbeiten möchten.

Sollten Sie also tatsächlich in einer Datenbankabfrage einen Join benötigen, empfiehlt sich eine Abfrage unter Umgehung des *Query Object Models*, auf die wir in Abschnitt IV.3.3 auf Seite 137 zu sprechen kommen.

IV.3.2.5 Mengenoperationen

Klassische Mengenoperationen, wie eine Vereinigung (engl. *union*), Schnittmenge (engl. *intersection*) und Differenzmengen (engl. *complement*), werden im *Query Object Model* nicht unterstützt (tatsächlich werden Schnitt- und Differenzmengen nicht einmal von MySQL unterstützt). Wir haben lediglich die Möglichkeit, diese Funktionen in PHP nachzubauen:

```
$union = array_merge($query1->execute(), $query2->execute());

$intersect = array_uintersect ( $query1->execute(), $query2->execute(),
  function($a, $b) { return $a->getUid() === $b->getUID() ? 1 : 0; } );

$complement = array_udiff ( $query1->execute(), $query2->execute(),
  function($a, $b) { return $a->getUid() === $b->getUid() ? 1 : 0; } );
```

IV.3.2.6 Gruppierung

Auch Gruppierungen nach unterschiedlichen Spalten werden vom QOM nicht unterstützt. Auch hier haben wir nur die Möglichkeit, die entsprechenden Funktionalitäten nachzubauen. Möchten wir zum Beispiel mit einer einzelnen Abfrage die Anzahl der Unterprojekte pro Projekt ermitteln, so würden wir normalerweise die SQL-Abfrage

```
SELECT uid, COUNT(1)
  FROM tx_mittwaldtimetrack_domain_model_project GROUP BY parent
```

verwenden. Stattdessen sind wir nun gezwungen, diese Berechnung in die Anwendung auszulagern:

```
$projects = $projectRepository->findAll();
ForEach($projects As $project) {
    $subProjectCount[$project->getUid()] = count($project->getChildren());
}
```

IV.3.3 Abfragen ohne das QOM

Wie wir mittlerweile gemerkt haben, steht mit dem *Query Object Model* zwar eine flexible Abstraktionsschnittstelle für die Datenbank zur Verfügung – und in der Tat sind wir nicht mehr an ein spezifisches Datenbankmanagementsystem wie MySQL oder PostgreSQL gebunden – dennoch sind wir einigen unliebsamen Einschränkungen unterworfen. Die fehlende Möglichkeit, Joins über mehrere Tabellen zu erstellen, wird zum Beispiel vielen Entwicklern sauer aufstoßen – den Verfasser eingeschlossen.

Der Grund für diese Einschränkungen liegt darin, dass Extbase zum größten Teil aus FLOW3 zurück portiert wurde. Im Gegensatz zu TYPO3 und Extbase setzt FLOW3 jedoch nicht auf einer klassischen relationalen Datenbank auf; sondern auf einem sogenannten *Content Repository* nach der JCR-Spezifikation. Dieses beschreibt eine von einem bestimmten Speichermedium (also zum Beispiel ein RDBMS, das Dateisystem, oder eine hierarchische XML-Datenbank) unabhängige Schnittstelle zum Speichern komplexer Objekte. Da dieses *Content Repository* nicht mit einzelnen Tabellen, sondern mit Objekten arbeitet, entfällt hier logischerweise die Notwendigkeit für Joins und Gruppierungen.

Für komplexere Abfragen, die über das QOM nicht abgebildet werden, können stellt dieses nun die Methode *statement* zur Verfügung, mit welcher wir direkt eine SQL-Abfrage übergeben können. Nachfolgend als Beispiel eine Methode aus dem *TimesetRepository*, um ohne Umweg über die *Assignment*-Klasse alle Zeitbuchungen für ein bestimmtes Projekt zu laden:

PHP: Classes/Domain/Repository/TimesetRepository.php

```
Class Tx_MittwaldTimetrack_Domain_Repository_TimesetRepository ↲
  Extends Tx_Extbase_Persistence_Repository {

    Public Function getTimesetsForProject ( ↲
      Tx_MittwaldTimetrack_Domain_Model_Project $project) {
```

```
        $extbaseFrameworkConfiguration = ↲
          Tx_Extbase_Dispatcher::getExtbaseFrameworkConfiguration();
        $pidList = implode(', ', t3lib_div::intExplode(',', ↲
          $extbaseFrameworkConfiguration['persistence']['storagePid']));

        $sql = ↲
          "SELECT t.*
           FROM     tx_mittwaldtimetrack_domain_model_timeset    t
               JOIN tx_mittwaldtimetrack_domain_model_assignment a ↲
                 ON t.assignment = a.uid
               JOIN tx_mittwaldtimetrack_domain_model_project    p ↲
                 ON a.project = p.uid
           WHERE     p.uid={$project->getUid()}
                 AND p.deleted + a.deleted + t.deleted = 0
                 AND p.pid IN ($pidList) AND a.pid IN ($pidList)
                 AND t.pid IN ($pidList)
           ORDER BY t.starttime DESC";

        $query = $this->createQuery();
        $query->statement($sql);
        Return $query->execute();

    }

}
```

Diese Methode bringt jedoch den großen Nachteil mit sich, dass die Datenbankunabhängigkeit verloren geht. Wir sind nun also davon abhängig, dass unsere Erweiterung auf einer relationalen Datenbank mit exakt den Tabellenstrukturen aufsetzt, wie sie in unserer SQL-Abfrage verwendet werden. Falls in der Abfrage sogar besondere Sprachelemente verwendet werden, geht unter Umständen sogar die Kompatibilität zu anderen relationalen Datenbankmanagementsystemen verloren. Ist dies für eine TYPO3-Erweiterung vielleicht noch in Ordnung – Datenbankabstraktion wird in den meisten Erweiterungen ja ohnehin eher stiefmütterlich behandelt, da die meisten Extensionentwickler einfach von MySQL als Datenbank ausgehen – so ist eine Portabilität nach FLOW3 in diesem Fall nicht mehr gewährleistet!

IV.4 Ein Blick unter die Persistenzschicht

Selbst bei intensiver Nutzung der verschiedenen Funktionen, die Extbase mit dem *Query Object Model* zur Verfügung stellt, erhält der Entwicklung in der Regel doch relativ wenig Einblick dahinter, wie genau Extbase eigentlich die verschiedenen Domänenobjekte dauerhaft in der Datenbank speichert. Diese Aufgabe übernimmt die *Persistenzschicht* (*persistence layer*) von Extbase, welche den Zugriff auf die Datenbank derart abstrahiert, dass der durchschnittliche Extension-Entwickler nicht dahinter zu schauen braucht.

Im folgenden Abschnitt soll einmal ein Blick unter die Haube der Persistenzschicht geworfen werden. Dazu betrachten wir zunächst den Vorgang des Data Mappings im Detail. Anschließend soll kurz erläutert werden, wie der Data Mapper nach eigenen Anforderungen konfiguriert werden kann.

IV.4.1 Data Mapping

Die Grundlagen des Data-Mappings wurden bereits in Kapitel 49 auf Seite III.1.4.1 angerissen. Im folgenden Abschnitt sollen die Vorgänge innerhalb des Data-Mappers noch einmal im Detail betrachtet werden.

Hauptaufgabe des Data-Mappers ist die objektrelationale Zuordnung; also die Zuordnung der Inhalte der Tabellenspalten auf die Attribute der PHP-Objekte. Hierfür wertet der DataMapper sowohl das *Table Configuration Array* aus, als auch die Kommentare innerhalb der Klassendefinitionen. Dabei wird eine Spalte nach den folgenden Kriterien analysiert (und zwar auch genau in dieser Reihenfolge):

1. Wird die Spalte auf ein Datum oder eine Uhrzeit validiert (enthält `config.eval` den Wert `date` oder `datetime`), so hat das entsprechende Klassenattribut automatisch den Typ `DateTime`.[16]
2. Ist die Spalte eine Checkbox (wenn `type` den Wert `check` hat), so wird der Spalteninhalt nach `boolean` (also entweder TRUE oder FALSE) konvertiert.
3. Wird die Spalte auf eine Ganzzahl validiert (enthält `config.eval` den Wert `int`), dann wird das Attribut auf den Datentyp `long` konvertiert.

16 Bei DateTime handelt es sich um eine PHP-Klasse, die seit Version 5.2 standardmäßig zur Verfügung steht. Siehe hierzu auch die offzielle PHP-Dokumentation unter http://de.php.net/datetime

4. Wenn die Spalte auf eine Fließkommazahl validiert wird (wenn `config.eval` also den Wert `double2` enthält), wird das Attribut nach double konvertiert.
5. Wenn eine Fremdtabelle angegeben ist (über die Eigenschaft `config.foreign_table`) und kein Fremdfeld (`config.foreign_field`), so wird davon ausgegangen, dass in der Spalte die UID eines anderen Objektes enthalten ist. Dem Klassenattribut wird dann dieses Objekt als Instanz der entsprechenden Klasse zugewiesen.
6. Wenn eine Fremdtabelle und ein Fremdfeld angegeben ist (sowohl `config.foreign_table` als auch `config.foreign_field`), so werden für dieses Attribut alle Datensätze aus der Fremdtabelle geladen, die eine Fremdschlüsse-Beziehung auf den aktuellen Datensatz haben.

IV.4.2 Zugriff auf fremde Datenquellen

Hin und wieder wird es passieren, dass aus einer Extbase-Erweiterung auf bereits bestehende Tabellen der TYPO3-Datenbank zugegriffen werden soll. In der Zeiterfassungs--Erweiterung aus dem ersten Kapitel wurde beispielsweise des öfteren auf die *fe_users*-Tabelle zugegriffen. Für diese bietet Extbase glücklicherweise bereits von Haus aus eine Schnittstelle an. Was aber, wenn wir auf eine Tabelle zugreifen möchten, für die Extbase keine Schnittstelle anbietet?

Nehmen wir als Beispiel einmal die *be_users*-Tabelle. Nun muss die Erweiterung zunächst einmal natürlich eine Klasse enthalten, auf welche die Tabelle abgebildet werden kann; an dieser Stelle heiße diese Klasse *Tx_MittwaldTimetrack_Domain_Model_BackendUser*. Die Zuordnung dieser Klasse zur *be_users*-Tabelle kann nun über das Typoscript-Setup der Erweiterung konfiguriert werden:

Typoscript: Configuration/TypoScript/setup.txt

```
plugin.tx_mittwaldtimetrack.persistence.classes {
    Tx_MittwaldTimetrack_Domain_Model_BackendUser {
        mapping {
            tableName = be_users
            recordType = Tx_MittwaldTimetrack_Domain_Model_BackendUser
            columns {
                email.mapOnProperty = emailAddress
            }
        }
    }
}
```

Sofern die Spalten der zugeordneten Tabelle dieselben Namen tragen wie die Klassenattribute, ist keine weitere Konfiguration notwendig; die Zuordnung der Datenbankspalten auf die Klassenattribute erfolgt vollautomatisch. Trägt die Spalte einen anderen Namen als das Attribut (oben beispielhaft an der Zuordnung der *email*-Spalte auf das *emailAddress*-Attribut verdeutlicht), muss dies explizit unter *columns* konfiguriert werden.

Damit diese Zuordnung funktioniert, muss die Ursprungstabelle ordnungsgemäß im TCA konfiguriert sein; außerdem müssen die im TCA konfigurierten Spaltentypen weiterhin mit den Datentypen der Klassenattribute übereinstimmen.

IV.4.3 Lazy- und Eager-Loading

Beim Wiederherstellen von Objekten aus der Datenbank stellt sich häufig die Frage, wie mit assoziierten Objekten zu verfahren ist. Grundsätzlich gibt es hier zwei verschiedene Verfahren, das sogenannte *Lazy Loading* und das *Eager Loading*.

Beim *Eager Loading* (dt. in etwa eifriges Laden) werden beim Laden eines Objektes aus der Datenbank alle assoziierten Objekte ebenfalls aus der Datenbank geladen. Wenn wir in der Zeiterfassungserweiterung also über das Projekt-Repository ein Projekt aus der Datenbank laden, würden beim Eager Loading sämtliche Projektmitgliedschaften sowie alle Zeitpaare und Benutzer für diese Mitgliedschaften ebenfalls aus der Datenbank geladen werden. Der Nachteil dieser Strategie liegt gerade bei großen Aggregaten darin, dass in der Regel eine große Menge an Daten geladen wird, von denen ein großer Teil in der Regel gar nicht benötigt wird.

Beim *Lazy Loading* (dt. in etwa faules Laden) hingegen werden Daten erst dann geladen, wenn sie tatsächlich benötigt werden. Extbase implementiert zu diesem Zweck die Klassen *Tx_Extbase_Persistence_LazyLoadingProxy* und *Tx_Extbase_Persistence_LazyObjectStorage*. Soll ein assoziiertes Objekt „faul“ geladen werden, so enthält das entsprechende Klassenattribut zunächst nur eine Instanz des LazyLoading-Proxys. Sobald irgendeine Methode dieses Attributes aufgerufen wird, lädt das LazyLoading-Objekt die entsprechenden Objekte aus der Datenbank und ersetzt sich selbst mit dem tatsächlichen Objekt:

```
Class Tx_MyExt_Domain_Model_Example Extends
Tx_Extbase_DomainObject_AbstractEntity {

        /**
         * @var Tx_MyExt_Domain_Model_OtherObject
         * @lazy
         */
```

```
    Protected $otherObject;

    Public Function testLazyLoading() {
        echo "Klassenname: ".get_class($this->otherObject);
          // Ausgabe: "Klassenname: Tx_Extbase_Persistence_LazyLoadingProxy"

        echo $this->otherObject->getUid();

        echo "Klassenname: ".get_class($this->otherObject);
          // Ausgabe: "Klassenname: Tx_MyExt_Domain_Model_OtherObject

    }

}
```

Wie genau dieser LazyLoading-Proxy nun implementiert ist, braucht uns an dieser Stelle nicht zu kümmern. Von Zeit zu Zeit kann es jedoch notwendig sein, das Laden des tatsächlichen Objektes zu „erzwingen“:

```
If($this->otherObject InstanceOf Tx_Extbase_Persistence_LazyLoadingProxy)
    $this->otherObject->_loadRealInstance();
```

IV.5 Fehlerbehandlung

Die Fehler- und Ausnahmebehandlung ist eine häufig unterschätzte Komponente. Obwohl Extbase und FLOW3 dem Entwickler hier bereits viel Arbeit abnehmen, zum Beispiel in Richtung Eingabevalidierung, kommen wir bei komplexeren Anwendungen nicht darum herum, der Fehlerbehandlung etwas mehr Aufmerksamkeit zukommen zu lassen.

IV.5.1 Validierung mit Validatorklassen

Gerade bei der Validierung von Benutzereingaben nimmt Extbase dem Entwickler bereits sehr viel Arbeit ab. So können wir zum Beispiel über einfache Anmerkungen in den Kommentaren Validierungsregeln für einzelne Attribute festlegen – wir erinnern uns an Kapitel III.4.5.2 auf Seite 96. Die Grenzen dieser eingebauten Validierung sind jedoch erreicht, wenn wir komplexere Objekte validieren wollen, deren Attribute miteinander in Zusammenhang stehen.

In der Zeiterfassungsanwendung aus dem letzten Kapitel konnte jeder Benutzer Zeitpaare für ein bestimmtes Projekt buchen. Jedes Zeitpaar zeichnete sich durch eine Anfangs- und eine Stoppzeit aus. Bisher wurden beide Zeiten jeweils getrennt auf Gültigkeit überprüft. Dies hinderte den Benutzer zwar daran, ein komplett ungültiges Datum anzugeben; jedoch kann es passieren, dass – auch wenn beide Zeiten jeweils für sich völlig korrekt sind – die Stoppzeit *vor* der Startzeit liegt.

Diesen Fall können wir mit der eingebauten Validierung nicht mehr abfangen, da wir hierfür mehr als ein Attribut auf einmal berücksichtigen müssen. Für solche komplexen Validierungen verwendet Extbase sogenannte Validatoren. Dies sind eigene Dienstklassen, die keinen anderen Zweck haben, als andere Objekte zu validieren. Dabei kann jedem Domänenobjekt eine eigene Validatorklasse zugeordnet werden.

Da diese Klassen als Dienst (wir erinnern uns an die verschiedenen Bestandteile der Anwendungsdomäne, die wir in Abschnitt II.1.2.3 auf Seite 25 besprochen hatten) im weitesten Sinne auch Bestandteil der Domäne sind, werden sie in dem Verzeichnis *Classes/Domain/Validator* gespeichert. Alle diese Klassen müssen Unterklassen der abstrakten Basisklasse *Tx_Extbase_Validation_Validator_AbstractValidator* sein. Die Benennung muss per Konvention mit dem zugehörigen Domänenobjekt übereinstimmen. Der Validator für ein Zeitpaar würde also den Namen *Tx_MittwaldTimetrack_Domain_Validator_TimesetValidator* tragen. Jede Validatorklasse muss die Methode *isValid* der Basisklasse überschreiben:

PHP: Classes/Domain/Validator/TimesetValidator.php

```
Class Tx_MittwaldTimetrack_Domain_Validator_TimesetValidator
 Extends Tx_Extbase_Validation_Validator_AbstractValidator {

    /**
     * @param Tx_MittwaldTimetrack_Domain_Model_Timeset $timeset
     */
    Public Function isValid($timeset) {

        If(!$timeset InstanceOf Tx_MittwaldTimetrack_Domain_Model_Timeset)
            $this->addError (
              Tx_Extbase_Utility_Localization::translate (
                'error-timeset-noTimeset', 'MittwaldTimetrack'),
              1265721022 );

        If($timeset->getStarttime() >= $timeset->getStoptime())
            $this->addError (
              Tx_Extbase_Utility_Localization::translate (
                'error-timeset-wrongTimeOrder', 'MittwaldTimetrack' ),
              1265721025 );

        Return count($this->getErrors()) === 0;
    }
}

?>
```

Versuchen wir nun, ein Zeitpaar mit zwei ungültigen Zeiten anzulegen, so erhalten wir folgende Fehlermeldung:

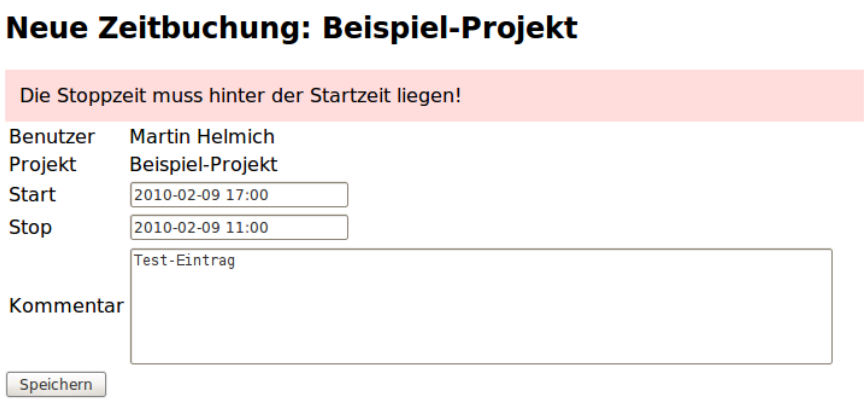

Abbildung 37: Der eigene Validator im Einsatz

IV.5.2 Ausnahmebehandlung

IV.5.2.1 Einführung in die strukturierte Ausnahmebehandlung

Von der Auswertung von Benutzereingaben einmal abgesehen, kann es auch im übrigen Programmablauf immer passieren, dass etwas Unerwartetes geschieht. Um in einem solchen Fall den regulären Programmfluss zu unterbrechen und geordnet zu beenden, werden in der Regel sogenannte *Ausnahmen* (engl. *exceptions*) verwendet. Wenn eine solche Bedingung eintritt, kann eine Methode eine solche Exception „werfen", welche dann an die nächsthöhere Methode weitergereicht wird. Diese Ausnahme wird so lange an die jeweils übergeordnete Methode weitergereicht, bis sie schließlich „aufgefangen" wird. In PHP werden hierfür die Schlüsselwörter *throw* zum Werfen und *catch* zum Auffangen einer Exception verwendet.

Codebereiche, aus denen eine Exception aufgefangen werden soll, werden von einem try-Block, gefolgt von einem *catch*-Befehl umfasst:

```
Try {
    doSomethingDangerous();
} Catch (Exception $e) {
    Echo "Ein Fehler ist aufgetreten: ".$e->getMessage();
    Exit(1);
}
```

IV.5.2.2 Werfen von Exceptions

In der Zeiterfassungserweiterung hatten wir beim Anlegen neuer Zeitpaare beispielsweise das Problem, dass wir überprüfen mussten, ob der aktuell eingeloggte Benutzer auch tatsächlich Mitglied des jeweiligen Projektes war. War dies nicht der Fall, hatten wir dort zunächst ein `Return '';` eingebaut. Zwar brach diese Anweisung den Programmfluss an besagter Stelle ab, allerdings wäre es in aller Regel schöner, in diesem Fall auch eine angemessene Fehlermeldung präsentiert zu bekommen. Des weiteren funktioniert diese Methode nur, solange ein Fehler direkt im Controller auftritt. Da der Großteil der Geschäftslogik jedoch im Domänenmodell untergebracht ist, ist die Wahrscheinlichkeit weitaus höher, dass ein Fehler innerhalb einer der Domänenklassen auftritt. Spätestens an dieser Stelle macht eine Fehlerbehandlung über Rückgabewerte nur noch begrenzt Sinn und verkompliziert die Anwendung unnötig.

Stattdessen könnten wir also auch einfach eine Exception werfen:

PHP: Classes/Controller/TimesetController.php

```
$user = $this->userRepository->findByUid (
  $GLOBALS['TSFE']->fe_user->user['uid'] );
$assignment = $user ? $project->getAssignmentForUser($user) : NULL;

If($assignment === NULL)
    Throw New Exception (
      "Sie sind kein Mitglied dieses Projektes!", 1266412844 );
```

Der Fehlercode 1266412844 wurde übrigens rein zufällig gewählt und dient lediglich dazu, einen bestimmten Ausnahmetyp eindeutig identifizieren zu können.

Und tatsächlich: Versuchen wir nun, ein Zeitpaar für ein Projekt zu buchen, in dem der aktuelle Benutzer kein Mitglied ist, so erhalten wir folgende Fehlermeldung:

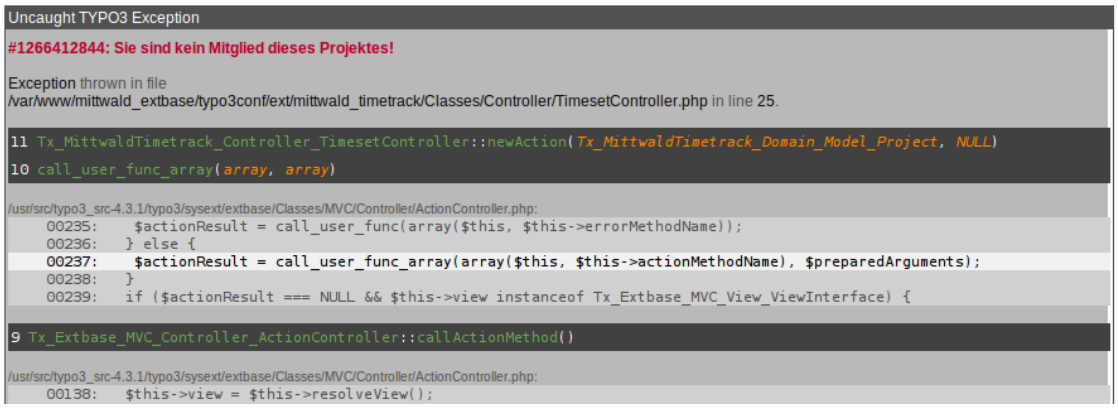

Abbildung 38: Eine Ausnahme wird vom DebugExceptionHandler aufgefangen

Was wir hier sehen, ist der im TYPO3-Core verankerte *DebugExceptionHandler*. Dieser zeigt zu jeder Ausnahme detaillierte Informationen über den Ursprung der Ausnahme an. Alternativ kann auch der sogenannte *ProductionExceptionHandler* verwendet werden, welcher eine minimalistischere Fehlermeldung anzeigt. Welcher von beiden verwendet wird, ist von der `displayErrors`-Einstellung im Installtool abhängig.

Nun ist solch eine Fehleranzeige natürlich auch nicht viel besser als gar keine Fehleranzeige. Optimal wäre nun natürlich eine Fehlermeldung, die dem Benutzer zwar die Fehlerursache ansprechend präsentiert, die aber trotzdem nicht das Seitenlayout zerstört. Zu diesen Zweck können wir nun einen eigenen *Exception Handler* implementieren.

IV.5.2.3 Ein eigener Exception Handler

IV.5.2.3.a Implementierung des Catch-Statements

Der TYPO3-interne *Exception Handler* ist dazu gedacht, alle Exceptions aufzufangen, die während des Seitenaufbaus geworfen werden. Möchten wir nun, dass die Ausnahmen aus unserer Erweiterung von einem eigenen Exception Handler bearbeitet werden, müssen wir diese „auffangen", bevor es der TYPO3-Exception Handler macht.

Zwar ist es möglich, mit der Konfigurationsvariablen *productionExceptionHandler* und *debugExceptionHandler* im Installations-Tool eigene Ausnahmebehandlungen zu definieren, allerdings haben all diese gemeinsam, dass durch eine Ausnahme das Seitenlayout zerstört wird. Wir müssen also eine Möglichkeit finden, die Exception früher abzufangen, vorzugsweise innerhalb unserer eigenen Erweiterung.

Zu diesem Zweck klinken wir uns in die Methode *callActionMethod* der *ActionController*-Klasse ein. Damit wir diese Methode nicht in jedem Controller wieder von neuem überschreiben müssen, erstellen wir eine abstrakte Controller-Klasse, die wir dann als Mutterklasse für den Projekt- und den Zeitpaar-Controller verwenden:

PHP: Classes/Controller/AbstractController.php

```
Abstract Class Tx_MittwaldTimetrack_Controller_AbstractController
  Extends Tx_Extbase_MVC_Controller_ActionController {

    Protected Function callActionMethod() {
        Try {
            parent::callActionMethod();
        } Catch(Exception $e) {
            $this->response->appendContent($e->getMessage());
        }
    }

}
```

Zur Erläuterung: Wir überschreiben die Methode *callActionMethod* der *ActionController*-Klasse. In dieser Methode rufen wir einfach dieselbe Methode der übergeordneten Klasse auf, umgeben den Aufruf aber mit einem *try-catch*-Statement. Auf diese Weise können wir alle Exceptions, die innerhalb irgendeiner der Controller-Aktionen geworfen werden, auffangen und bearbeiten.

Im Projekt- und im Zeitpaar-Controller passen wir nun die Klassendefinition entsprechend an, so dass diese nun nicht mehr die *Tx_Extbase_MVC_Controller_ActionController*-Klasse, sondern die neue *Tx_MittwaldTimetrack_Controller_AbstractController*-Klasse erweitern.

Versuchen wir nun noch einmal, ein Zeitpaar in einem Projekt zu buchen, in dem wir kein Mitglied sind, erscheint nun immer noch die Fehlermeldung, dieses Mal aber in den Seitenaufbau integriert.

IV.5.2.3.b Verschönerung der Ausgabe

Ziel des nächsten Abschnittes soll es nun sein, die Fehlerausgabe noch ein wenig zu verschönern. Ideal wäre es, wenn auch die Ausgabe eines Fehlers über ein Fluid-Template geregelt werden könnte. Damit dies jedoch auch für mehrere Controller funktioniert, müssen wir ein wenig tricksen:

PHP: Classes/Controller/AbstractController.php

```
Protected Function handleError(Exception $e) {
    $controllerContext = $this->buildControllerContext();
    $controllerContext->getRequest()->setControllerName('Default');
    $controllerContext->getRequest()->setControllerActionName('error');
    $this->view->setControllerContext($controllerContext);

    $content = $this->view->assign('exception', $e)->render('error');
    $this->response->appendContent($content);
}

Protected Function callActionMethod() {
    Try {
        parent::callActionMethod();
    } Catch(Exception $e) {
        $this->handleError($e);
    }
}
```

Die handleError-Methode gaukelt dem View-Objekt vor, dass es sich in einem Controller namens „Default“ befindet (dieser Controller muss nicht existieren). Vorteil dieser Methode ist, dass für mehrere Controller nur ein Template für die Fehlerbehandlung verwendet werden muss. Anschließend wird die aufgefangene Ausnahme dem View zur Darstellung zugewiesen.

Das Fehler-Template kann dann mit allen Möglichkeiten gestaltet werden, die von Fluid zur Verfügung gestellt werden:

HTML: Resources/Templates/Default/error.html

```
<h2>Fehler!</h2>

<div style="background-color: #fdd; padding: 12px;">
  <div style="font-weight: bold;">Fehler #{exception.code}!</div>
  <div>{exception.message}</div>
  <div style="color: #666; font-size: 10px; margin: 12px;">
    <f:format.nl2br>{exception.traceAsString}</f:format.nl2br>
  </div>
</div>
```

Betrachten wir nun noch einmal unsere Erweiterung im Frontend, so wird das soeben erstelle Fluid-Template zur Darstellung der Fehlermeldung verwendet:

Abbildung 39: Verwendung eines Fluid-Templates zur Darstellung von Exceptions

IV.6 Backend-Module

Das Erstellen eines Backendmoduls hat sich mit Extbase enorm vereinfacht. Prinzipiell unterscheidet sich die Erstellung eines Backendmoduls nicht von dem Anlegen eines Frontend-Plugins. Zunächst muss das neue Modul in der *ext_tables.php* registriert werden. Dabei muss eine Liste von Controller-Aktionen angegeben werden, die über das Backendmodul angesprochen werden dürfen. Des weiteren können wir angeben, wo das neue Modul im Backend-Menü positioniert werden soll:

PHP: ext_tables.php

```
If (TYPO3_MODE === 'BE')  {

    Tx_Extbase_Utility_Extension::registerModule(
      $_EXTKEY,                                  # Extension-Key
      'web',                                     # Kategorie
      'tx_mittwaldtimetrack_m1',                 # Modulname
      '',                                        # Position
      Array ( 'Backend' => 'index' ),            # Controller
      Array ( 'access'  => 'user,group',         # Konfiguration
              'icon'    => 'EXT:'.$_EXTKEY.'/ext_icon.gif',
              'labels'  => 'LLL:EXT:' . $_EXTKEY
                 . '/Resources/Private/Language/locallang_mod.xml' )
    );

}
```

In der Regel macht es Sinn, für das Backendmodul einen neuen Controller anzulegen. Grundsätzlich können aber auch dieselben Controller wie für das Frontendplugin verwendet werden.

Derzeit befindet sich die Extbase-Schnittstelle für Backendmodule noch im experimentellen Status. Aus diesem Grund werden wir im folgenden Abschnitt zwar die grundlegenden Konzepte betrachten, das Thema jedoch nicht sonderlich vertiefen, da sich in der Zukunft mit großer Wahrscheinlichkeit noch mehrere Schnittstellen wieder ändern werden.

Erstellen wir zunächst auf gewohnte Weise einen neuen Controller für das Backendmodul:

PHP: Classes/Controller/BackendController.php

```
Class Tx_MittwaldTimetrack_Controller_BackendController
  Extends Tx_Extbase_MVC_Controller_ActionController {

    Public Function indexAction() {
```

```
        $this->view->assign('foo', 'bar')
                   ->assign('bar', 'baz');
    }

}
```

Im View können wir auf einige spezielle View-Helper zurück greifen, die Fluid extra für die Verwendung in Backendmodulen zur Verfügung stellt:

HTML: Resources/Private/Templates/Backend/Index.html

```
<f:be.container pageTitle="Timetracking">
  <div style="float:right">
    <f:be.menus.actionMenu>
      <f:be.menus.actionMenuItem label="Overview" controller="Backend"
                                 action="index" />
      <f:be.menus.actionMenuItem label="Aktion Eins" controller="Backend"
                                 action="action1" />
      <f:be.menus.actionMenuItem label="Aktion Zwei" controller="Backend"
                                 action="action2" />
    </f:be.menus.actionMenu>
  </div>
  <div>
    <f:be.pageInfo /> -
    <f:be.pagePath />
  </div>

  <h1>Projekte</h1>

  <f:be.tableList tableName="tx_mittwaldtimetrack_domain_model_project"
                  fieldList="{0: 'name', 1: 'parent'}"
                  storagePid="16" levels="2" recordsPerPage="10"
                  sortField="name" sortDescending="true" readOnly="true"
                  enableClickMenu="false" clickTitleMode="info"
                  alternateBackgroundColors="true" />

</f:be.container>
```

Die meisten der Backend-ViewHelper, die von Fluid zur Verfügung gestellt werden, greifen auf die interne TYPO3-API zur Darstellung von Backend-Komponenten zurück. Auf diese Weise ist auch in Extbase-Backendmodulen möglich, denselben *Look and Feel* bereitzustellen, wie im übrigen TYPO3-Backend.

Abbildung 40: Ein Extbase-Backendmodul

IV.7 Hinter den Kulissen: Der Dispatcher

Nachdem wir in den letzten Abschnitten einige der fortgeschritteneren Möglichkeiten betrachtet haben, die Extbase bietet, soll der nun folgende Abschnitt vornehmlich einen Blick hinter die Kulissen des Frameworks werfen. Die hier vermittelten Kenntnisse sind zum Entwickeln von Erweiterungen mit Extbase zwar nicht unbedingt notwendig, erleichtern jedoch das grundlegende Verständnis der Funktions- und Verhaltensweisen von Extbase.

Die zentrale Komponente von Extbase ist der sogenannte *Dispatcher* (engl. etwa für *Disponent* oder *Verteiler*). Dieser wird durch die PHP-Klasse *Tx_Extbase_Dispatcher* abgebildet. Wird nun über die Methode *Tx_Extbase_Utility_Extension::registerPlugin* ein neues Plugin registriert, so wird nun jeder Aufruf an dieses Plugin nicht – wie bei einer pi_base-Erweiterung – direkt an die Erweiterung geleitet, sondern zunächst an den Extbase-Dispatcher.

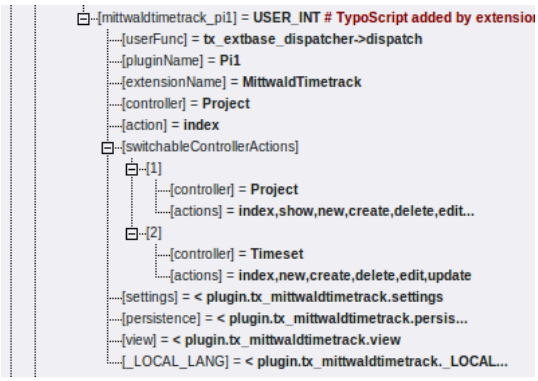

Abbildung 41: Konfiguration des Extbase-Dispatchers

Das Plugin wird dabei zunächst als USER_INT-Objekt erstellt. Als `userFunc`-Attribut ist die Methode *Tx_Extbase_Dispatcher->dispatch* registriert. Der Dispatcher erstellt nun ein *Request*-Objekt (eine Instanz der Klasse *Tx_Extbase_MVC_Web_Request*), welches unter anderem die `$_GET`- und `$_POST`-Parameter enthält. Aus diesen Parametern versucht der Dispatcher nun diejenige Controller-/Action-Kombination zu ermitteln, die diesen *Request* am besten bearbeiten kann.

Wurde ein passender Controller gefunden, so wird dieser instanziert und der neuen Instanz das *Request*-Objekt zur Bearbeitung übergeben. Der Controller erstellt dann in aller Regel ein View-Objekt und ruft die entsprechende Action-Methode auf. Aus dem Ergebnis dieser Action-Methode erstellt der Controller ein *Response*-Objekt (eine Instanz der Klasse *Tx_Extbase_MVC_Web_Response*), welches unter anderem den HTML-Inhalt enthält, welcher von dem View generiert wurde.

Dieses *Response*-Objekt wird zurück an den Dispatcher übermittelt, welches den *Response* auswertet und anschließend den generierten HTML-Inhalt zurückliefert.

Abbildung 42: Funktionsweise des Extbase-Dispatchers

Über die Methode *callModule* ist der Dispatcher auch für die Darstellung von Backendmodulen zuständig.

Teil V: Anhang

V.1 Referenz: Fluid-ViewHelper

V.1.1 Allgemeine

V.1.1.1 Alias

Der Alias-ViewHelper erstellt eine neue Variable innerhalb des Fluid-Templates. Die Variablen sind nur innerhalb dieses ViewHelpers gültig.

```
<f:alias map="{foo : 'bar'}">
  Foo hat den Wert "{foo}".
</f:alias>
```

Ausgabe des Beispiels:

```
Foo hat den Wert "bar"
```

V.1.1.2 cObject

Dieser ViewHelper stellt ein Typoscript-Inhaltsobjekt dar. Der Typoscript-Pfad wird in dem Attribut `typoscriptObjectPath` angegeben. Optional kann dem Objekt über das Attribut `data` spezieller Inhalt zugewiesen werden.

```
<f:cObject typoscriptObjectPath="plugin.tx_myext.object" data="{object}" />
```

V.1.1.3 Count

Der *Count*-ViewHelper zählt die Anzahl der Elemente in einem Array oder einer Liste.

```
Der Array beinhaltet <f:count subject="{0:'eins', 1:'zwei', 2:'drei'}" />
Objekte.
```

Ausgabe des Beispiels:

```
Der Array beinhaltet 3 Objekte.
```

V.1.1.4 Cycle

Dieser ViewHelper rotiert durch eine Liste von Werten. Dieser ViewHelper kann zum Beispiel verwendet werden, um alternierende Zeilen in Tabellen zu erzeugen.

```
<ul>
  <f:for each="{0:1, 1:2, 2:3, 3:4}" as="foo">
    <f:cycle values="{0: 'odd', 1: 'even'}" as="zebraClass">
      <li class="{zebraClass}">{foo}</li>
    </f:cycle>
  </f:for>
</ul>
```

Ausgabe des Beispiels:

```
<ul>
  <li class="odd">1</li>
  <li class="even">2</li>
  <li class="odd">3</li>
  <li class="even">4</li>
</ul>
```

V.1.1.5 Image

Stellt ein Bild innerhalb des Templates dar und skaliert es gegebenenfalls auf eine bestimmte Größe. Neben dem *src*-Attribut werden folgende Parameter akzeptiert:

- *width* und *height*: Geben die Größe an, in der das Bild dargestellt werden soll
- *minWidth* und *minHeight*: Geben die Größe an, in der das Bild mindestens dargestellt werden soll. Ist das Bild kleiner, wird es proportional vergrößert.
- *maxWidth* und *maxHeight*: Die Größe, in der das Bild höchstens dargestellt werden darf. Ist das Bild größer, wird es proportional verkleinert.
- *alt* und *title*: Legen die üblichen Alternativtexte und Beschriftungen für dieses Bild fest.
- Universelle Attribute: Attribute wie *class* zum Festlegen einer CSS-Klasse, *style* für Inline-CSS oder *id* werden an den erstellten Image-Tag weitergereicht.

Beispiel:

```
<f:image src="path/to/image.png" maxWidth="100" />
```

V.1.1.6 Translate

Der *Translate*-ViewHelper liefert einen sprachabhängigen Text zurück, der standardmäßig aus der *locallang.xml* ausgelesen wird. Dieser ViewHelper akzeptiert die folgenden Parameter:

- *key*: Der Index, unter dem der sprachabhängige Text gespeichert ist. Optional kann diese Angabe einen Dateinamen enthalten. Wenn kein Dateiname angegeben ist, wird der Text in der *locallang.xml* der Erweiterung gesucht.
- *default* (optional): Der Standardwert. Dieser wird verwendet, wenn für den angegebenen Index in der aktuellen Sprache kein Text gefunden wurde.
- *htmlEscape* (optional): Wenn dieser Wert gesetzt ist, werden Sonderzeichen in dem zurückgelieferten Text durch ihre entsprechenden HTML-Codierungen ersetzt. Dieser Wert ist standardmäßig aktiviert.
- *arguments* (optional): Die Werte in diesem Array werden automatisch in den Text eingefügt, wenn dieser Platzhalter im Stil von `%s` enthält.

Beispiele:

```
<f:translate key="my_ll_key" />
Ausgabe: Mein Text

<f:translate key="my_nonexisting_ll_key" default="Mein anderer Text" />
Ausgabe: Mein anderer Text

<f:translate key="my_special_text" />
Ausgabe: Mein &lt;b&gt;besonderer&lt;/b&gt; Text

<f:translate key="my_special_text" htmlEscape="false" />
Ausgabe: Mein <b>besonderer</b> Text

<f:translate key="my_dynamic_text" />
Ausgabe: Mein %s Text

<f:translate key="my_dynamic_text" arguments="{'spezieller, dynamischer'}" />
Ausgabe: Mein spezieller, dynamischer Text
```

V.1.2 Kontrollstrukturen

V.1.2.1 If

Der *If*-ViewHelper stellt den Inhalt nur dann dar, wenn die Bedingung, welche im Attribut *condition* angegeben wurde, erfüllt ist. Die Bedingung kann entweder ein einfacher Wahrheitswert sein oder auch ein logischer Ausdruck.

```
<f:if condition="{object.someProperty}">
    Die Bedingung ist wahr!
</f:if>
```

```
<f:if condition="{object.someOtherProperty} == 5}">
    Die Eigenschaft "someOtherProperty" hat den Wert 5!
</f:if>
```

V.1.2.2 Then/Else

Der *Then*- und der *Else*-ViewHelper dienen zur Erweiterung des *If*-ViewHelpers. Der Inhalt des *Then*-ViewHelpers wird angezeigt, wenn die Bedingung des umschließenden *If*-ViewHelpers erfüllt ist; der Inhalt des Else-ViewHelpers wird angezeigt, wenn die Bedingung nicht erfüllt ist.

```
<f:if condition="{value} == 5">
  <f:then>{value} ist 5</f:then>
  <f:else>{value} ist ungleich 5</f:else>
</f:if>
```

V.1.2.3 For

Der *For*-ViewHelper bietet die Möglichkeit, alle Werte in einem Array in einer Schleife durchlaufen zu können. Dabei wird mit dem *each*-Attribut das zu durchlaufende Array übergeben, das *as*-Attribut beschreibt den Namen, unter dem die einzelnen Elemente innerhalb der Schleife zur Verfügung stehen sollen. Optional kann auch das *key*-Attribut angegeben werden. Dieses bezeichnet den Namen, unter welchem jeweils der Index jedes Elementes abgebildet werden soll. Mit dem reverse-Attribut schließlich kann die Reihenfolge der Elemente vertauscht werden.

```
<f:for each="{objects}" as="object">{object.value} </f:for>
Ausgabe: Wert1 Wert2 Wert3

<f:for each="{objects}" as="object" key="key">{key}: {object.value} </f:for>
Ausgabe: 0: Wert1 1: Wert2 2: Wert3

<f:for each="{objects}" as="object" reverse="true">{object.value} </f:for>
Ausgabe: Wert3 Wert2 Wert1
```

V.1.2.4 GroupedFor

Der *GroupedFor*-ViewHelper bietet die Möglichkeit, ein Array von Objekten nach einem bestimmten Attribut zu gruppieren. Beispiel (übernommen aus der offiziellen Fluid-API-Dokumentation):

```
<f:groupedFor each="{0: {name: 'Apfel', color: 'Grün'},
                     1: {name: 'Kirsche', color: 'Rot'},
                     2: {name: 'Banane', color: 'Gelb'},
```

```
                    3: {name: 'Erdbeere', color: 'Rot'}}"
              as="fruitsOfThisColor"
              groupBy="color"
              groupKey="color">
  Obst mit der Farbe {color}:
  <f:for each="{fruitsOfThisColor}" as="fruit">{fruit.name} </f:for>
</f:groupedFor>
</ul>
```

Ausgabe des Beispiels:

```
Obst mit der Farbe Grün: Apfel
Obst mit der Farbe Rot: Kirsche Erdbeere
Obst mit der Farbe Gelb: Banane
```

V.1.3 Formulare

V.1.3.1 Form

Der Form-ViewHelper ist Grundlage eines jeden Formulars. Dieser ViewHelper akzeptiert die folgenden Parameter:

- *extensionName*, *pluginName*, *controller* und *action*: Eine Extension-/Controller-/Action-Kombination, zu welcher dieses Formular beim Abschicken weitergeleitet werden soll. *extensionName*, *pluginName* und *controller* sind optional.
- *arguments*: Zusätzliche Argumente, die an die Zielaktion übergeben werden sollen.
- *object* und *name*: Mit *object* kann dem Formular ein Domänenobjekt zugewiesen werden. Wenn ein Objekt zugewiesen wurde, kann der Formularname entfallen.
- *method*: Wie bei normalen Formular sind hier „get“ und „post“ mögliche Werte.
- *onsubmit* und *onreset*: Javascript-Code, der beim Zurücksetzen bzw. Abschicken des Formulars durchgeführt werden soll.

Beispiel:

```
<f:form object="{myObject}" action="create" controller="Default" method="get"
onsubmit="alert('Hallo!');">

    <!-- ... -->

</f:form>
```

V.1.3.2 Formularelemente allgemein

Sämtliche Formular-ViewHelper, die von Fluid zur Verfügung gestellt werden, akzeptieren die folgenden Parameter:

- *property*: Mit diesem Parameter kann dem Eingabefeld ein bestimmtes Attribut eines Domänenobjektes zugewiesen werden. Selbstverständlich ist dies nur dann möglich, wenn dem übergeordneten Formular über das *object*-Attribut ein Objekt zugeordnet wurde.
- *name* und *value*: Diese Attribute bezeichnen den Namen und den Inhalt des Formularfeldes. Diese Werte müssen nur angegeben werden, wenn dem Formularfeld kein Objektattribut zugeordnet ist.

V.1.3.3 Form.Checkbox

Stellt ein einzelnes Auswahlfeld dar. Zusätzlich zu den üblichen Attributen wird hier noch das *checked*-Attribut ausgewertet.

```
<f:form.checkbox name="checkMe" checked="{myValue} == 1" /> Klick mich!

Ausgabe:
  <input type="checkbox" name="tx_myext_pi1[checkMe]" checked="checked" />
  Klick mich!
```

V.1.3.4 Form.Hidden

Der *Hidden*-ViewHelper stellt ein verstecktes Formularelement dar:

```
<f:form.hidden name="dontLookAtMe" value="Hi there!" />

Ausgabe:
  <input type="hidden" name="tx_myext_pi1[dontLookAtMe]" value="Hi there!" />
```

V.1.3.5 Form.Password und Form.Textbox

Diese beiden ViewHelper stellen einzeilige Text-Eingabefelder dar. Die Besonderheit bei dem Passwort-Eingabefeld ist, dass die einzelnen Zeichen lediglich verdeckt dargestellt werden. Zusätzlich zu den üblichen Parametern akzeptieren beide ViewHelper noch die folgenden:

- *disabled*: Deaktiviert das Eingabefeld

- *readonly*: Das Eingabefeld bleibt zwar aktiv, kann aber dennoch nicht mehr bearbeitet werden.
- *size* und *maxlength*: Beschreiben die Größe und die maximale Eingabelänge des Feldes.

```
Passwort: <f:form.password name="password" size="30" maxlength="30" />

Ausgabe:
  Passwort: <input type="password" name="password" size="30" maxlength="30" />
```

V.1.3.6 Form.Radio

Stellt einen Radio-Button dar. Der Unterschied von Radiobuttons im Gegensatz zu Checkboxen besteht darin, dass von mehreren Radio-Buttons mit gleichem Namen jeweils nur eine ausgewählt werden kann. Zusätzlich zu den üblichen Parametern werden die Parameter *checked* und *disabled* akzeptiert. Wenn das Eingabefeld einem Objektattribut zugeordnet ist, muss der *checked*-Parameter nicht angegeben werden.

```
Lieblings-Fastfood:
<f:form.radio property="fastfood" value="0" /> McDonald's
<f:form.radio property="fastfood" value="1" /> Burger King

Ausgabe:
  <input type="radio" name="tx_myext_pi1[obj][fastfood]" value="0" />
    McDonald's
  <input type="radio" name="tx_myext_pi1[obj][fastfood]" value="1" />
    Burger King
```

V.1.3.7 Form.Select

Dieser ViewHelper stellt ein Mehrfach-Auswahlfeld dar. Die zur Auswahl stehenden Optionen werden als Array über den *options*-Parameter übergeben. Handelt es sich bei diesen Optionen nicht um einfache Strings, sondern um Domänenobjekte, kann mit dem Parameter *optionLabelField* angegeben werden, welches Attribut der Objekte für die Beschriftung der Auswahloptionen verwendet werden soll.

Zusätzlich werden außerdem die folgenden Parameter akzeptiert:

- *multiple*: Falls aktiv, können mehrere Optionen auf einmal ausgewählt werden.
- *size*: Die Größe des Auswahlfeldes.

```
Lieblings-Bier:
```

```
<f:form.select property="favoriteBeer"
               options="{0: 'Becks', 1: 'Barre Bräu', 2: 'Herforder'}" />

Ausgabe:
  <select name="tx_myext_pi1[objectname][favoriteBeer]">
    <option value="0">Becks</option>
    <option value="1">Barre Bräu</option>
    <option value="2">Herforder</option>
  </select>
```

V.1.3.8 Form.Textarea

Der *Textarea*-ViewHelper stellt ein mehrzeiliges Eingabefeld dar. Die Attribute dieses ViewHelpers entsprechen in etwa denen des tatsächlichen *Textarea*-HTML-Elementes:

- *rows* und *cols*: Geben die Größe des Eingabefeldes in Zeilen und Spalten an. Diese beiden Parameter müssen auf jeden Fall gesetzt werden.
- *disabled*: Mit diesem Parameter kann das Eingabefeld deaktiviert werden.

```
<f:form.textarea property="motto" rows="5" cols="40" />

Ausgabe:
  <textarea name="tx_myext_pi1[objectname][motto]" rows="5" cols="40">
    Wer später bremst, fährt länger schnell.
  </textarea>
```

V.1.3.9 Form.Submit

Dieser ViewHelper stellt eine Schaltfläche zum Abschicken der Formulardaten bereit.

```
<f:form.submit value="Speichern!" />
```

V.1.4 Formatierung

V.1.4.1 Format.Crop

Der *Crop*-ViewHelper schneidet einen Text auf eine bestimmte Länge zurecht. Die folgenden Parameter können zur Konfiguration verwendet werden:

- *maxCharacters*: Die Anzahl der Zeichen, auf die der Text beschnitten werden soll.

- *append*: Eine Zeichenkette, die an den Text an gehangen werden soll, falls dieser wirklich gekürzt wurde. Wenn dieser Parameter nicht angegeben wurde, wird standardmäßig „…“ an gehangen.
- *respectWordBoundaries*: Wenn dieser Parameter gesetzt ist – was standardmäßig der Fall ist – wird der Text nicht inmitten eines Wortes abgeschnitten.

```
<f:format.crop maxCharacters="20">
  Hallo! Das hier ist ein sehr langer Text!</f:format.crop>
Ausgabe: Hallo! Das hier ist...

<f:format.crop maxCharacters="20" append=" und so weiter.">
  Hallo! Das hier ist ein sehr langer Text!</f:format.crop>
Ausgabe: Hallo! Das hier ist und so weiter.

<f:format.crop maxCharacters="20" respectWordBoundaries="false">
  Hallo! Das hier ist ein sehr langer Text!</f:format.crop>
Ausgabe: Hallo! Das hier ist ei...
```

V.1.4.2 Format.Currency

Dieser ViewHelper formatiert eine Zahl als Währungsangabe. Dabei wird die Zahl auf die zweite Nachkommastelle aufgerundet. Die Darstellung der Währung wird durch die folgenden Parameter bestimmt:

- *currencySign*: Das zu verwendende Währungszeichen. Wird dieses Attribut nicht angegeben, wird wird kein Währungszeichen verwendet.
- *decimalSeparator*: Das Zeichen, das als Dezimaltrennzeichen verwendet werden soll. Dies ist standardmäßig das in Europa übliche Komma.
- *thousandsSeperator*: Das Zeichen, welches zur Trennung von Tausenderstellen verwendet werden soll. Dies ist standardmäßig der Punkt.

```
<f:format.currency>12345.152</f:format.curreny>
Ausgabe: 12.345,15

<f:format.currency currencySign="€" thousandsSeparator=" ">
  12345.152</f:format.currency>
Ausgabe: 12 345,15 €

<f:format.currency currencySign="$" decimalSeparator="."
                   thousandsSeparator=",">12345.152</f:format.currency>
Ausgabe: 12,345.15 $
```

V.1.4.3 Format.Date

Der *Date*-ViewHelper dient zum Formatieren von Kalenderdaten. Folgende Parameter werden akzeptiert:

- *date*: Das darzustellende Datum. Wenn kein Datum angegeben wird, wird der Inhalt des ViewHelpers als Datum verwendet. Dieses Datum kann entweder eine Instanz der *DateTime*-Klasse oder eine Zeichenkette sein.
- *format*: Dieser Parameter beschreibt, wie das Datum zu formatieren ist. Siehe hierzu auch die Dokumentation zur PHP-Funktion *date*.[17] Wenn kein Format angegeben ist, wird das Datum im Format Y-m-d formatiert.

```
<f:format.date>1266506951</f:format.date>
Ausgabe: 2010-02-18

<f:format.date date="{object.tstamp}" format="r" />
Ausgabe: Thu, 18 Feb 2010 16:30:00 +0100
```

V.1.4.4 Format.Html

Dieser ViewHelper dient zur Ausgabe von HTML-Inhalt, der zum Beispiel im TYPO3-Backend mithilfe des RTE erstellt wurde. Dazu wird eine sogenannte *parseFunc* verwendet, die über Typoscript konfiguriert werden kann (standardmäßig unter `lib.parseFunc_RTE`). Dieser Typoscript-Pfad kann mit dem Parameter *parseFuncTSPath* überschrieben werden.

```
<f:format.html>
  Hallo! Das <b>hier</b> ist ein <link 123>Link</link>.</f:format.html>

Ausgabe: Hallo! Das <b>hier</b> ist ein <a href="index.php?id=123">link</a>.
```

V.1.4.5 Format.Nl2br

Der *Nl2br*-ViewHelper konvertiert normale Zeilenumbrüche (genauer gesagt den sogenannten Zeilenvorschub) in HTML-Zeilenumbrüche, die mit `<br />` codiert werden.

```
<format.nl2br>
  Hallo Welt!
  Nächste Zeile!</format.nl2br>
```

17 Siehe http://php.net/date

```
Ausgabe:
  Hallo Welt!<br />
  Nächste Zeile!
```

V.1.4.6 Format.Number

Dieser ViewHelper dient – ähnlich dem *Currency*-ViewHelper – dem Formatieren von Zahlen. Zur Angabe der Nachkommastellen steht hier der *decimals*-Parameter zur Verfügung. Ansonsten entspricht dieser ViewHelper dem *Currency*-ViewHelper.

```
<f:format.number decimals="1" decimalSeparator="," thousandsSeparator=" ">
  12345.67</f:format>
Ausgabe: 12 345,6
```

V.1.4.7 Format.Printf

Der *Printf*-ViewHelper formatiert Zeichenketten mit dem *printf*-Befehl. Die Parameter, die in den String eingefügt werden sollen, können über das *arguments*-Attribut übergeben werden:

```
<f:format.printf argumemts="{0: 12345.6789}">
  Die Zahl hat den Wert %.2f</f:format.printf>
```

V.1.5 Links

V.1.5.1 Link.Action

Dieser ViewHelper erstellt einen Link zu einer anderen Action-/Controller-Kombination. Dieser ViewHelper kann über die folgenden Parameter konfiguriert werden:

- *extensionName*, *pluginName*, *controller* und *action*: Mit diesen Parametern kann eine Controller-Aktion eindeutig identifiziert werden. Wenn eine dieser Angaben weggelassen wird, wird stattdessen der aktuelle Controller/Extension/… für den Link verwendet.
- *arguments*: Zusätzliche Argumente, die in den Link eingebaut werden sollen.
- *pageUid* und *pageType*: Die UID und der Typ der Seite, auf die der Link zeigen soll. Werden diese Parameter nicht angegeben, wird auf die aktuelle Seite verlinkt.
- *noCache* und *noCacheHash*: Steuert das Caching-Verhalten der verlinkten Seite.

- *section*: Gibt den *Fragmentbezeichner* der URL an, also zum Beispiel einen Anker innerhalb des Dokuments.
- *format*: Das Dateiformat, in welchem die Ziel-URL erstellt werden soll (also zum Beispiel „.html“
- *linkAccessRestrictedPages*: Wenn dieser Parameter gesetzt ist, werden auch Links zu Seiten dargestellt, auf die der Benutzer keinen Zugriff hat. Dies ist standardmäßig nicht der Fall.
- *additionalParams*: Zusätzliche Argumente für die URL. Im Gegensatz zu dem *arguments*-Parameter, wird diesen nicht der Extensionname als Präfix vorangestellt.
- *absolute*: Ist dieser Parameter gesetzt, wird eine absolute URL erstellt.
- *addQueryString*: Durch Setzen dieses Parameters werden der *Query String* (der Teil der URL ab dem „?“) der aktuellen URL in die URL des neuen Links übernommen.
- *argumentsToBeExcludedFromQueryString*: Dieses Argument gibt Parameter an, die nicht aus dem aktuellen *Query String* übernommen werden sollen.

V.1.5.2 Link.Email

Erstellt einen Link zu einer E-Mail-Adresse. Entsprechend der Frontend-Konfiguration wird der E-Mail-Link gegebenenfalls verschlüsselt dargestellt.

```
<f:link.email email="email@invalid.tld" />
Ausgabe: <a href="mailto:email@invalid.tld">email@invalid.tld</a>
```

V.1.5.3 Link.External

Erstellt einen Link zu einer externen URL.

```
<f:link.external uri="http://www.mittwald.de">Klick mich</f:link.external>
Ausgabe: <a href="http://www.mittwald.de">Klick mich</a>
```

V.1.5.4 Link.Page

Dieser ViewHelper erstellt einen Link zu einer anderen Seite aus dem TYPO3-Seitenbaum. Es werden dieselbe Parameter akzeptiert wir bei dem Link.Action-ViewHelper (bis auf die Parameter *extensionName*, *pluginName*, *controller* und *action*).

V.2 Quelltexte

Die in der Dokumentation verwendeten Quelltexte sowie die erstellte Extension stehen unter http://www.mittwald.de/extbase-dokumentation/ zum Download zur Verfügung

Teil VI: Literatur und Quellen

1. Bartonitz, Martin: *Content Repository – Einheitlicher Zugriff auf elektronische Daten in Datenbanken*. In: WissenHeute, Ausgabe 10/2009. Telekom Training, Münster.
2. Evans, Eric: *Domain Driven Design – Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003. ISBN 978-0-32112-521-7.
3. Kurfürst, Sebastian: *Fluid: Templating leicht gemacht – Neue Template-Engine für FLOW3 und TYPO3 4.x*. In t3n Nr. 16, 5/2009. Yeebase media-Verlag, Hannover.
4. Kurfürst, Sebastian: *Fluid Templating System – Manual*. Abgerufen 11. 2. 2010, URL: http://flow3.typo3.org/documentation/manuals/fluid/
5. Lahres, Bernhard; Raýman, Gregor: *Objektorientierung – Professionelle Entwurfsverfahren*. Galileo Computing, 2006. ISBN 978-3-89842-624-4.
6. Lemke, Robert: *Domain-Driven Design – Lösungsansätze für die Entwicklung komplexer Webanwendungen*. In: t3n Nr. 11, 2/2008. Yeebase media-Verlag, Hannover.
7. Lemke, Robert; Dambekalns, Karsten: *FLOW3 Framework – Manual*. Abgerufen 11. 2. 2010, URL: http://flow3.typo3.org/documentation/manuals/flow3/
8. Michaelsen, Sebastian: *Extbase-Extensions im Handumdrehen*. In: t3n Nr. 21, 9—11/2010. Yeebase media-Verlag, Hannover.
9. Rau, Jochen; Kurfürst, Sebastian: *Zukunftssichere TYPO3-Extensions mit Extbase und Fluid – Der Einstieg in die neue Extension-Entwicklung*. O'Reilly 2010. ISBN 978-3-89721-965-6

10. Rau, Jochen; Kurfürst, Sebastian: *Extbase Extension Programming*. Revision 1844, 25. 10. 2009. https://svn.typo3.org/TYPO3v4/CoreProjects/MVC/doc_extbase/Documentation/

11. Rau, Jochen: *Neues MVC-Framework „Extbase“ ebnet den Weg von 4.x zu FLOW3*. In t3n Nr. 16, 5/2009. Yeebase media-Verlag, Hannover.