

Einstieg in das Zend Framework

Von Rob Allen, www.akrobat.com
Übersetzung von Daniel Messer, blog.selfphp.org
Dokument Revision 1.0.3
Copyright © 2006

Das Zend Framework ist zur Zeit in der Version 0.1.5, also wird es Zeit, mich ranzusetzen und ein Einstiegs-Tutorial zu schreiben! Dieser Guide ist dafür gedacht, eine sehr einfache Einleitung in das Zend Framework zu geben um eine sehr einfache, datenbank-getriebene Anwendung zu schreiben.

HINWEIS: Dieses Tutorial ist für die Version 0.1.5 des Zend Frameworks gedacht und wird wahrscheinlich nicht ohne weiteres mit dem neusten Code aus dem Subversion-Repository des Projekts oder neueren Versionen laufen!

Model-View-Controller Architektur

Der gewöhnliche Weg, eine PHP-Anwendung zu schreiben sieht in etwa so aus:

```
<?php
include "common-libs.php";
include "config.php";
mysql_connect($hostname, $username, $password);
mysql_select_db($database);
?>

<?php include "header.php"; ?>
<h1>Home Page</h1>

<?php
$sql = "SELECT * FROM news";
$result = mysql_query($sql);
?>
<table>
<?php
while ($row = mysql_fetch_assoc($result)) {
?>
<tr>
    <td><?php echo $row['date_created']; ?></td>
    <td><?php echo $row['title']; ?></td>
</tr>
<?php
}
?>
</table>
<?php include "footer.php"; ?>
```

Im Laufe ihres Lebenszyklus, wird eine Anwendung dieser Art schwer wartbar, da vom Kunden gewünschte Änderung über die gesamte Code-Basis verstreut eingefügt werden.

Eine Methode die Wartung und Betreuung einer Web-Applikation zu erleichtern, ist die Heraustrennung des Codes einer Web-Seite in 3 klar getrennte Aufgabenbereiche (und somit meistens in getrennte Dateien):

Model	Das „Model“ beschreibt den Teil einer Anwendung, der sich mit den Eigenschaften der Daten beschäftigt, die verarbeitet werden sollen. In dem obigen Beispiel ist das das Konzept von „News“. Dadurch konzentriert sich das „Model“ im Allgemeinen immer auf die Fachlogik einer Anwendung und kümmert sich meistens um das Laden und Speichern von Daten aus einer Datenbank.
View	Die „View“ besteht aus den Teilen einer Anwendung, die sich mit der Anzeige der Daten aus dem „Model“ für den Benutzer beschäftigen. Typischerweise generiert eine „View“ dabei HTML.
Controller	Ein „Controller“ verbindet „Model“ und „View“ und stellt sicher, dass die richtigen Daten korrekt auf der Seite angezeigt werden.

Das Zend Framework entspricht einer solche Model-View-Controller Architektur (kurz: MVC). Sie wird benutzt um die gleichzeitige Entwicklung an verschiedenen Teilen einer Anwendung und spätere Wartung zu vereinfachen.

Voraussetzungen

Das Zend Framework hat folgende Voraussetzungen:

- PHP 5.1.4 (oder neuer)
- Apache Web-Server mit mod_rewrite

Das Framework beziehen

Das Framework kann unter <http://framework.zend.com/download> entweder als .zip oder als .tar.gz heruntergeladen werden. Zum Zeitpunkt der Entstehung dieses Tutorials ist Version 0.1.5 aktuell. Die Anwendungen, die wir in diesem Tutorial schreiben werden, funktioniert nur mit dieser Version!

Verzeichnisstruktur

Während das Framework selbst keine bestimmte Verzeichnisstruktur verlangt, empfiehlt die Dokumentation eine gleich bleibende Verzeichnisorganisation im gesamten Projekt. Diese nimmt an, dass du volle administrative Kontrolle über den Apache Webserver hast. Da wir uns das Leben jedoch ein wenig erleichtern wollen, nehmen wir eine kleine Modifikation gegenüber der offiziellen Empfehlung vor.

Lege zunächst ein Verzeichnis namens `zf-tutorial` im *DocumentRoot* des Apache an. Das bedeutet, dass die URL der Applikation (*Anmerkung des Übs.:* bei einer lokalen Installation des Apache) <http://localhost/zf-tutorial> lauten wird.

Erzeuge dort dann die folgende Verzeichnishierarchie:

```
zf-tutorial/  
  /application  
    /controllers  
    /models  
    /views  
  /library  
  /public  
    /images  
    /scripts  
    /styles
```

Wie du sehen kannst, haben wir für Model-, View- und Controller-Skripte jeweils getrennte Verzeichnisse geschaffen. Bilder, JavaScript- und CSS-Dateien werden ebenfalls in eigenen

Verzeichnissen unter `/public` gehalten. Das heruntergeladene Zend Framework wird im Verzeichnis `/library` gespeichert. Andere eventuell benötigte Bibliotheken kommen auch hier rein.

Entpacke das heruntergeladene Archiv - in meinem Fall `ZendFramework-0.1.5.zip` in ein temporäres Verzeichnis. Beim Entpacken entsteht dort ein Verzeichnis namens `ZendFramework-0.1.5`. Kopiere nun den Inhalt von `ZendFramework-0.1.5/library` nach `zf-tutorial/library`. Letzteres sollte danach ein Verzeichnis namens `zend` und eine Datei namens `zend.php` aufweisen.

Bootstrapping

Der Controller des Zend Framework, wurde dafür konzipiert, saubere URLs für Webseiten zu ermöglichen. Um das zu erreichen, müssen zunächst alle Aufrufe an eine einzelne `index.php` weitergeleitet werden. Das Verfahren entspricht dem [Bootstrapping](#), daher wird diese `index.php` auch „Bootstrapper“ genannt. Er liefert uns einen zentralen Einstiegspunkt für die Applikation und stellt sicher, dass die Laufzeitumgebung für die gesamte Anwendung richtig und einheitlich konfiguriert ist. Für die Realisierung verwenden wir eine verzeichnisbasierte Konfigurationsdatei des Apache, die standardmäßig `.htaccess` heißt und in `zf-tutorial/` liegen wird.

Ihr Inhalt muss wie folgt aussehen:

zf-tutorial/.htaccess

```
RewriteEngine on
RewriteRule .* index.php

php_flag magic_quotes_gpc off
php_flag register_globals off
```

Die Redirect-Regel in Zeile 2 ist sehr einfach und bedeutet so viel wie: „leite alles nach `index.php` um“.

Des Weiteren haben wir einige `php.ini`-Einstellungen zwecks Sicherheit vorgenommen. Diese sollten eigentlich schon als Standard so eingestellt sein, doch wir wollen hier auf Nummer Sicher gehen. Beachte, dass die Verwendung der Apache-Direktive `php_flag` durch `mod_php` bereitgestellt wird, sie funktionieren also nur, wenn PHP als Modul in den Indianer eingebunden wird. Wenn du eine CGI/FastCGI-Version von PHP benutzt, nimm bitte die Einstellungen direkt in der `php.ini` vor.

Die obige Rewrite-Konfiguration leitet tatsächlich alle Aufrufe unterhalb von `zf-tutorial` weiter. Bei Bildern oder Stylesheets ist das jedoch nicht gewünscht. Da wir solche Dateien jedoch alle unter `public/` halten, lässt sich die Konfiguration des Apache für dieses Verzeichnis bequem ändern:

zf-tutorial/public/.htaccess

```
RewriteEngine off
```

Wir würden jedoch gerne auch unsere `application/`- und `library/`-Verzeichnisse gegen direkte Zugriffe schützen, dafür benötigen wir ein paar weitere `.htaccess`-Dateien:

zf-tutorial/application/.htaccess

```
deny from all
```

zf-tutorial/library/.htaccess

```
deny from all
```

Beachte bei dem Einsatz von `.htaccess`-Dateien, dass die Direktive „AllowOverride“ in der `httpd.conf` auf „All“ gesetzt werden muss, damit die Anweisungen in den `.htaccess`' akzeptiert werden. Die Idee für den kreativen Einsatz dieser Dateien stammt übrigens von Jayson Minard's Artikel "[Blueprint for PHP Applications: Bootstrapping \(Part 2\)](#)". Ich kann euch die gesamte Artikel-Reihe wärmstens empfehlen!

Die Bootstrap-Datei: index.php

zf-tutorial/index.php ist unser „Bootstrapper“, wir beginnen daher mit dem folgenden Code:

zf-tutorial/index.php

```
<?php
error_reporting(E_ALL|E_STRICT);
date_default_timezone_set('Europe/London');

set_include_path('.' . PATH_SEPARATOR . './library/'
    . PATH_SEPARATOR . './application/models');
include "Zend.php";

Zend::loadClass('Zend_Controller_Front');
Zend::loadClass('Zend_Controller_RewriteRouter');

// setup controller
$route = new Zend_Controller_RewriteRouter();
$route->addRoute('edit', ':controller/:action/id/:id',
    array('controller' => 'index', 'action' => 'index'));
$controller = Zend_Controller_Front::getInstance();
$controller->setRouter($route);

// run!
$controller->run('./application/controllers');
```

Beachte, dass wir kein abschließendes `?>` gesetzt haben, da es nicht benötigt wird und das Weglassen Seiteneffekte von sonst schwer zu findenden Fehler verhindert.

Lass uns nun die Datei Schritt für Schritt durchgehen.

```
error_reporting(E_ALL|E_STRICT);
date_default_timezone_set('Europe/Berlin');
```

Diese Zeilen sorgen dafür, dass wir alle Fehler sehen, die wir machen (vorausgesetzt in der `php.ini` steht „display_errors“ auf „on“). Außerdem setzen wir noch unsere aktuelle Zeitzone, wie PHP 5.1 das neuerdings benötigt.

```
set_include_path('.' . PATH_SEPARATOR . './library/'
    . PATH_SEPARATOR . './application/models');
include "Zend.php";
```

Das Zend Framework ist so aufgebaut, dass es seine Dateien über den include-Pfad erreichen können muss. Wir haben ebenfalls unser `model`-Verzeichnis hinzugefügt, damit wir später bequem unsere Model-Klassen laden können, ohne immer den vollen Pfad in der Importanweisung angeben zu müssen. Um endlich loszulegen, müssen wir noch die `Zend.php` inkludieren, um Zugriff auf die `Zend` Klasse zu erhalten, die die benötigten, statischen Methoden besitzt um beliebige andere Zend Framework Klassen zu laden.

```
Zend::loadClass('Zend_Controller_Front');
Zend::loadClass('Zend_Controller_RewriteRouter');
```

`Zend::loadClass` lädt und inkludiert die im Argument benannte Klasse. Das wird erreicht, indem die die Unterstriche durch Verzeichnistrenner (unter Unix `/`, unter Windows `\`) ersetzt und `.php` ans Ende angehängen wird. Dadurch wird die Klasse `Zend_Controller_Front` aus der Datei `Zend/Controller/Front.php` geladen. Wenn du dieselben Namenskonventionen in deinen eigenen Bibliotheken verfolgst, kannst du die `loadClass()`-Methode ebenfalls anwenden, um Klassen von dort einzubinden.

Als nächstes richten wir den Controller ein und fügen eine Weiterleitung hinzu, die es erlaubt, eine ID mit der URL mitzuschicken. Standardmäßig erlaubt der `Zend_Controller_RewriteRouter` Weiterleitungen nur für „/controller/action“. Wir benötigen jedoch eine Weiterleitung für „/controller/action/id/{x}“, sonst landet der User direkt auf dem Controller und nicht auf dem Bootstrapper. Das wird übrigens ab Version 0.1.6 nicht mehr nötig sein, dann ist auch diese Weiterleitung per Default eingerichtet.

```
// setup controller
$route = new Zend_Controller_RewriteRouter();
$route->addRoute('edit', ':controller/:action/id/:id',
    array('controller' => 'index', 'action' => 'index'));
$controller = Zend_Controller_Front::getInstance();
$controller->setRouter($route);
```

Und dann weiter zum Knackpunkt der Anwendung: Wir erhalten eine Instanz vom Zend Framework Controller, richten den zuvor erzeugten Router auf ihm ein und rufen anschließend Startmethode `run()` auf. Beachte, dass `run()` den Pfad zu dem Verzeichnis benötigt, in dem unsere eigenen Controller liegen werden.

```
// run!
$controller->run('./application/controllers');
```

Falls du jetzt auf den Gedanken kommst, <http://localhost/zf-tutorial> aufzurufen, wirst du einen „Fatal Error“ ähnlich diesem erhalten:

```
Fatal error: Uncaught exception 'Zend_Exception' with message 'File
"./application/controllers/IndexController.php" was not found.
(etc.)
```

Er sagt uns, dass wir unsere Anwendung noch nicht vollständig eingerichtet haben, da er den `IndexController` nicht finden kann. Bevor wir das tun, sollten wir uns aber erst einmal ein paar Gedanken machen, was unsere Anwendung eigentlich können soll. Lass uns das also als nächstes tun.

Die Anwendung

Wir werden ein sehr einfaches CD-Datenbanksystem aufbauen. Die Hauptseite wird unsere vorhandene CD-Kollektion auflisten und Links zum Bearbeiten, Hinzufügen oder Löschen von CDs bereitstellen. Die Begriffe „CD“ und „Album“ sollen im Folgenden synonym behandelt werden. Wir werden unsere Sammlung in einer Datenbank speichern, die eine Tabelle mit folgendem Schema enthält:

<i>Feldname</i>	<i>Typ</i>	<i>Null?</i>	<i>Besonderheiten</i>
id	Integer	Nein	Primärschlüssel, Auto-Increment
artist	Varchar(100)	Nein	
title	Varchar(100)	Nein	

Erforderliche Seiten

Es werden folgende Seiten entstehen:

Homepage	Diese Anzeige präsentiert die Liste aller Alben und wird Links zum Bearbeiten, Löschen und Hinzufügen solcher bereitstellen.
Album hinzufügen	Diese Seite wird ein Formular zum Hinzufügen eines Albums anzeigen.
Album bearbeiten	Diese Seite wird ein vorausgefülltes Formular zum Editieren eines Albums bereitstellen.
Album löschen	Diese Seite wird eine Bestätigung für den Löschvorgang verlangen und anschließend das Album löschen.

Die Seiten organisieren

Bevor wir unsere Seiten erstellen, müssen wir verstehen, in welcher Organisation das Framework sie erwartet. Jede Seite ist eine so genannte „Action“ und solche „Actions“ werden in Controller gruppiert. Zum Beispiel ist für die URL <http://localhost/zf-tutorial/news/view> „news“ der Controller und „view“ die Action. Das erlaubt das einfache Gruppieren von verwandten Actions. Ein Controller „news“ könnte neben „view“ noch weitere Actions haben, zum Beispiel „current“ oder „archived“.

Das Standardverhalten des Zend Frameworks bezüglich eines beliebigen Controllers reserviert bereits eine spezielle Action „index“ als Standard-Action. Das heißt, für eine URL wie <http://localhost/zf-tutorial/news/> wird die Action „index“ innerhalb des Controllers „news“ automatisch ausgeführt. Ebenfalls erwartet das Framework einen Standard-Controller, wenn keiner angegeben wurde. Dieser heißt – welche Überraschung - auch „index“. Folglich wird beim Aufruf von <http://localhost/zf-tutorial/> die Action „index“ im Controller „index“ ausgeführt.

Da dies sein sehr einfaches Tutorial sein soll, werden wir uns hier nicht mit „komplizierteren“ Sachen wie einem Anmeldesystem herumschlagen. Das können wir uns für ein separates Tutorial aufheben...

Zurück zum Thema. Wir haben vier Seiten, die alle etwas mit Alben zu tun haben. Daher gruppieren wir sie alle als Actions in einen gemeinsamen Controller. Am besten wir benutzen den Default-Controller:

<i>Page</i>	<i>Controller</i>	<i>Action</i>
Homepage	Index	Index
Album hinzufügen	Index	Add
Album bearbeiten	Index	Edit
Album löschen	Index	Delete

Schön einfach, oder?

Den Controller einrichten

Wir haben nun alles zusammen, um unseren Controller zu erstellen. Im Zend Framework ist ein Controller eine Klasse, die wie folgt heißen muss: {Name des Controller}Controller. Beachte, dass der Name des Controllers mit einem Großbuchstaben beginnen muss. Der Dateiname, in dem diese Klasse steht, muss {Name des Controller}Controller.php lauten und sie muss in dem von uns im include()-Pfad angegebenen Verzeichnis(sen) liegen. Wieder beginnt hier der Name des Controllers im Dateinamen mit einem Großbuchstaben, der Rest des Namens wird klein geschrieben. Jede Action ist eine öffentliche Methode (Sichtbarkeit: `public`) des Controllers, die nach dem Schema {Name der Action}Action() benannt wird. Hier wiederum muss der Name der Action mit Kleinbuchstaben beginnen.

Folglich heißt unser Controller `IndexController`, der in `zf-tutorial/application/controllers/IndexController.php` definiert ist.

zf-tutorial/application/controllers/IndexController.php

```
<?php

class IndexController extends Zend_Controller_Action
{
    function indexAction()
    {
        echo "<p>in IndexController::indexAction()</p>";
    }

    function addAction()
    {
        echo "<p>in IndexController::addAction()</p>";
    }
}
```

```

function editAction()
{
    echo "<p>in IndexController::editAction()</p>";
}

function deleteAction()
{
    echo "<p>in IndexController::deleteAction()</p>";
}
}

```

Fürs Erste haben wir den Controller so eingerichtet, dass jede Action ihren Namen ausgibt. Teste das ruhig, in dem du über die folgenden URLs navigierst:

URL	Ausgabe
http://localhost/zf_tutorial/	in IndexController::indexAction()
http://localhost/zf_tutorial/index/add	in IndexController::addAction()
http://localhost/zf_tutorial/index/edit	in IndexController::editAction()
http://localhost/zf_tutorial/index/delete	in IndexController::deleteAction()

Wie wir sehen funktioniert zumindest das Default-Routing korrekt und die richtigen Actions werden für die jeweilige Seite ausgeführt.

Nun ist es an der Zeit, die „View“ zu bauen.

Die View einrichten

Die View-Komponente im Zend Framework heißt – irgendwie wenig überraschend – `Zend_View`. Sie erlaubt uns die Trennung des Codes, der das HTML generiert von dem Code, der in den Action-Komponenten enthalten ist.

Die allgemeine Benutzung sieht in etwa so aus:

```

$view = new Zend_View();
$view->setScriptPath('/path/to/view_files');
echo $view->render('view.php');

```

Es stellt sich schnell heraus, dass, wenn wir diese Zeilen in jede unserer 4 Action-Funktionen packen würden, wir uns unnötig oft wiederholen und außerdem den eigentlichen Code mit Sachen belasten, die dort nichts zu suchen haben.

Viel lieber würden wir unsere View-Komponente woanders definieren und dann in den Actions auf das bereits initialisierte Objekt zugreifen.

Die Entwickler des Zend Frameworks haben dieses Problem vorausgesehen und stellen uns eine Art Registry für das Speichern und Laden von Objekten zur Verfügung. Die Anweisung, um ein Objekt dort zu registrieren lautet:

```

Zend::register('obj', $object);

```

Um das gespeicherte Objekt wieder zu erhalten, rufen wir einfach die folgende Anweisung auf:

```

$object = Zend::registry('obj')

```

Um unsere View-Komponente nun in die Anwendung zu integrieren, initialisieren wir sie und fügen sie gleich im Bootstrapper (`zf-tutorial/index.php`) der Registry hinzu.

Der relevante Teil der zf-tutorial/index.php

```
...
Zend::loadClass('Zend_Controller_Front');
Zend::loadClass('Zend_Controller_RewriteRouter');
Zend::loadClass('Zend_View');

// register the view we are going to use
$view = new Zend_View();
$view->setScriptPath('./application/views');
Zend::register('view', $view);

// setup controller
$route = new Zend_Controller_RewriteRouter();
$controller = Zend_Controller_Front::getInstance();
...

```

Die Änderungen an der Datei sind in Fettschrift gekennzeichnet und sollten selbsterklärend sein. Beachte, dass wir die Klasse `Zend_View` zuerst via `Zend::loadClass()` laden, und dann den Script-Pfad zu dem Verzeichnis unserer View-Objekte setzen müssen.

Das registrierte View-Objekt können wir nun in unseren Action-Methoden benutzen und den Code zur Testausgabe so in eine View-Komponente verschieben.

Ändere dazu den `IndexController` wie folgt. Die Änderungen sind wieder fett geschrieben.

zf-tutorial/application/controllers/IndexController.php

```
<?php

class IndexController extends Zend_Controller_Action
{
    function indexAction()
    {
        $view = Zend::registry('view');
        $view->title = "My Albums";
        echo $view->render('indexIndex.tpl.php');
    }

    function addAction()
    {
        $view = Zend::registry('view');
        $view->title = "Add New Album";
        echo $view->render('indexAdd.tpl.php');
    }

    function editAction()
    {
        $view = Zend::registry('view');
        $view->title = "Edit Album";
        echo $view->render('indexEdit.tpl.php');
    }

    function deleteAction()
    {
        $view = Zend::registry('view');
        $view->title = "Delete Album";
        echo $view->render('indexDelete.tpl.php');
    }
}

```

In jeder Methode holen wir uns das View-Objekt aus der Registry, weisen ihr eine Eigenschaft „title“ zu und zeigen dann das entsprechende Template an (rendern). Es sollte klar sein, dass wir 4 dieser Templates benötigen. Als Konvention habe ich jedes Template nach seinem Controller und nach seiner Action benannt und als Dateiendung „.tpl.php“ verwendet, um sie von normalen PHP-Skripten unterscheiden zu können. Es sind trotzdem nichts weiter als PHP-Skripte und nicht etwa HTML-Dateien mit einer bestimmten Template-Syntax.

zf-tutorial/application/views/indexIndex.tpl.php

```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <h1><?php echo $this->escape($this->title); ?></h1>
</body>
</html>
```

zf-tutorial/application/views/indexAdd.tpl.php

```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <h1><?php echo $this->escape($this->title); ?></h1>
</body>
</html>
```

zf-tutorial/application/views/indexEdit.tpl.php

```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <h1><?php echo $this->escape($this->title); ?></h1>
</body>
</html>
```

zf-tutorial/application/views/indexDelete.tpl.php

```
<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <h1><?php echo $this->escape($this->title); ?></h1>
</body>
</html>
```

Beim Testen sollte jede Action ihren Namen als HTML-Überschriftelement anzeigen.

Gemeinsamer HTML-Code

Bei der Entwicklung des HTML-Codes wird sehr schnell auffallen, dass eine Anwendung oft auf mehreren Seiten den gleichen HTML-Code verwendet. Wir entfernen aus allen Templates den gemeinsamen Code und lagern ihn in einem einzelnen Template aus, das wir `site.tpl.php` nennen. Das können wir benutzen, um einen gemeinsamen Rahmen für jede Seite vorzugeben, zu der jede Action dann nur noch ihren spezifischen Teil hinzufügt. Die Funktionalität dafür verankern wir im PHP-Code der `site.tpl.php`. Unser Controller benötigt wieder ein paar Änderungen:

zf-tutorial/application/controllers/IndexController.php

```
<?php

class IndexController extends Zend_Controller_Action
{
    function indexAction()
    {
        $view = Zend::registry('view');
        $view->title = "My Albums";
        $view->actionTemplate = 'indexIndex.tpl.php';
        echo $view->render('site.tpl.php');
    }

    function addAction()
    {
```

```

        $view = Zend::registry('view');
        $view->title = "Add New Album";
        $view->actionTemplate = 'indexAdd.tpl.php';
        echo $view->render('site.tpl.php');
    }

function editAction()
{
    $view = Zend::registry('view');
    $view->title = "Edit Album";
    $view->actionTemplate = 'indexEdit.tpl.php';
    echo $view->render('site.tpl.php');
}

function deleteAction()
{
    $view = Zend::registry('view');
    $view->title = "Delete Album";
    $view->actionTemplate = 'indexDelete.tpl.php';
    echo $view->render('site.tpl.php');
}
}

```

Wir haben die Template-Variablen „actionTemplate“ eingeführt, in der wir den Namen des Templates speichern, dass die Rahmenseite letztendlich mit Inhalt füllt und rendern nun in allen Fällen site.tpl.php. Die Templates sehen nun wie folgt aus:

zf-tutorial/application/views/site.tpl.php

```

<html>
<head>
    <title><?php echo $this->escape($this->title); ?></title>
</head>
<body>
    <div id="content">
        <?php echo $this->render($this->actionTemplate); ?>
    </div>
</body>
</html>

```

zf-tutorial/application/views/indexIndex.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
```

zf-tutorial/application/views/indexAdd.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
```

zf-tutorial/application/views/indexEdit.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
```

zf-tutorial/application/views/indexDelete.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
```

Anmerkung des Übersetzers: Es mag Einigen vielleicht merkwürdig vorkommen, wie wir dem View-Objekt Template-Variablen zuweisen. Woher soll die Zend_View alle unsere Variablennamen kennen? Das braucht sie nicht. Die in PHP 5 überarbeitete Objektorientierung ermöglicht es Objekten, über die speziellen Methoden `__get($varName)` und `__set($varName, $varValue)` zur Laufzeit dynamisch Objektvariablen hinzugefügt zu bekommen. Die beiden Methoden implementieren jeweils das Holen des Inhalts einer Variablen anhand ihres Namens beziehungsweise das Setzen/Überschreiben einer Variablen anhand ihres Namens mit einem gegebenen Wert. Unterstützt werden nur skalare Werte. Man spricht hierbei auch vom „Überladen“ eines Objektes.

Styling

Auch wenn das hier nur ein Tutorial ist, werden wir eine gemeinsame CSS-Datei schreiben, um unsere Anwendung etwas ansehlicher zu gestalten.

zf-tutorial/application/views/site.tpl.php

```
...
<head>
    <title><?php echo $this->escape($this->title); ?></title>
    <link rel="stylesheet" type="text/css" media="screen"
        href="/zf-tutorial/public/styles/site.css" />
</head>
...
```

zf-tutorial/public/styles/site.css

```
body,html {
    font-size:100%;
    margin: 0;
    font-family: Verdana,Arial,Helvetica,sans-serif;
    color: #000;
    background-color: #fff;
}

h1 {
    font-size:1.4em;
    color: #000080;
    background-color: transparent;
}

#content {
    width: 770px;
    margin: 0 auto;
}

label {
    width: 100px;
    display: block;
    float: left;
}

#formbutton {
    margin-left: 100px;
}
```

Die Datenbank

Nun, da wir in unserer Anwendung den Code zur Steuerung von dem der Anzeigelogik getrennt haben, ist es Zeit uns dem Abschnitt „Model“ zuzuwenden. Zur Erinnerung: Der Model-Part beschäftigt sich mit dem Hauptgegenstand unserer Applikation (den so genannten „Business-Rules“), in unserem Fall also mit der Datenbank, die die Alben speichert. Wir werden von der `Zend_Db_Table`-Klasse des Frameworks Gebrauch machen, mit der das Suchen, Einfügen, Modifizieren oder Löschen von Datensätzen in einer Tabelle abgewickelt wird.

Konfiguration

Bevor wir `Zend_Db_Table` benutzen können, müssen wir der Klasse zunächst sagen, welche Datenbank sie in Verbindung mit welchen Benutzernamen und Passwort benutzen soll. Weil wir diese Informationen nicht fest in unserem Code verankern wollen, lagern wir sie in einer Konfigurationsdatei aus.

Das Framework stellt die Klasse `Zend_Config` bereit, mit der sich flexibel und objekt-orientiert auf die Konfiguration zugreifen lässt. Als Quellformate werden derzeit PHP-Arrays, INI- und XML-Dateien unterstützt.

Wir werden die folgende INI-Datei benutzen:

zf-tutorial/application/config.ini

```
[general]
db.adapter = PDO_MYSQL
db.config.host = localhost
db.config.username = benutzername
db.config.password = passwort
db.config.dbname = datenbankname
```

Natürlich solltest du hier deine eigenen Daten eintragen.

Die Benutzung von `Zend_Config` ist sehr einfach:

```
$cfg_array = Zend_Config_Ini::load('config.ini', 'section');
$config = new Zend_Config($cfg_array);
```

Beachte, dass `Zend_Config_Ini` nur einen Abschnitt der INI-Datei lädt, nicht alle. Es unterstützt jedoch ein spezielles Schlüsselwort, `extends`, um weitere Abschnitte zu laden. Außerdem nutzt es den Punkt in Schlüsseln in der INI-Datei, um eine hierarchische Gruppierung von verwandten Daten zu ermöglichen. In unserer `config.ini` werden der Host, der Benutzername, das Passwort und der Datenbankname unter `$config->db->config` gruppiert.

Wir laden unsere Konfigurationsdatei gleich im Bootstrapper:

Der relevante Teil zf-tutorial/index.php

```
...
Zend::loadClass('Zend_Controller_Front');
Zend::loadClass('Zend_Controller_RewriteRouter');
Zend::loadClass('Zend_View');
Zend::loadClass('Zend_Config');
Zend::loadClass('Zend_Config_Ini');

// load configuration
$config = new Zend_Config(Zend_Config_Ini::load(
    './application/config.ini', 'general'));
Zend::register('config', $config);

// register the view we are going to use
$view = new Zend_View();
$view->setScriptPath('./application/views');
...

```

Die Änderungen sind wie immer in Fettschrift hervorgehoben. Wir laden die beiden Klassen, mit denen wir arbeiten werden und laden anschließend die Einstellungen aus dem Abschnitt „general“ aus unserer Konfigurationsdatei `application/config.ini` in `$config`.

Beachte, dass in unserem Fall die Variable `$config` eigentlich nicht in der Registry speichern müssten, da wir sie hier nur im Bootstrapper benötigen. Aber gewöhne es dir besser an, denn in richtigen Anwendungen wirst du in der Konfigurationsdatei sicherlich mehr als nur Datenbankzugangsdaten speichern.

Zend_Db_Table benutzen

`Zend_Db_Table` müssen wir zunächst die Zugangsdaten unserer Datenbank übergeben, die wir eben geladen haben. Dafür erzeugen wir zunächst eine Instanz von `Zend_Db` und registrieren sie über die statische Methode `Zend_Db_Table::registerDefaultAdapter()`. Wir tun das wieder gleich im Bootstrapper. Änderungen sind wieder in Fettschrift gekennzeichnet.

Ausschnitt von zf-tutorial/index.php

```
...
Zend::loadClass('Zend_Config');
Zend::loadClass('Zend_Config_Ini');
Zend::loadClass('Zend_Db');
Zend::loadClass('Zend_Db_Table');

// load configuration
$config = new Zend_Config(Zend_Config_Ini::load(
    './application/config.ini', 'general'));
Zend::register('config', $config);

// setup database
$db = Zend_Db::factory($config->db->adapter,
    $config->db->config->asArray());
Zend_Db_Table::setDefaultAdapter($db);

// register the view we are going to use
$view = new Zend_View();
$view->setScriptPath('./application/views');
Zend::register('view', $view);
...
```

Die Tabelle erzeugen

Ich werde MySQL benutzen, daher lautet das SQL-Statement um unsere Tabelle zu erzeugen:

```
CREATE TABLE album (
    id int(11) NOT NULL auto_increment,
    artist varchar(100) NOT NULL,
    title varchar(100) NOT NULL,
    PRIMARY KEY (id)
)
```

Du kannst diese Anweisung entweder direkt auf der Konsole mit dem MySQL-Kommandozeilen-Client oder auf einer Weboberfläche wie phpMyAdmin ausführen.

Testdaten erzeugen

Wir werden auch ein paar Zeilen in die Tabelle eintragen, um die Abfragefunktionalität unserer Anwendung zu testen. Ich habe hier die ersten 2 Alben der „Top 100“ von amazon.de genommen:

```
INSERT INTO album (artist, title)
VALUES
('Es ist wie es ist', 'Pur'),
('Rudebox', 'Robbie Williams');
```

Das Model

`Zend_Db_Table` ist eine abstrakte Basisklasse für jede Tabelle, also müssen wir sie ableiten und eine unserer gewünschten Funktionalität entsprechende, spezifische Implementierung schreiben. Standardmäßig sieht die Klasse vor, dass die zu verwaltende Tabelle genau so heißt wie die Klasse. Daher nennen wir sie `Album`, damit sie die Tabelle `album` verwaltet. `Zend_Db_Table` nimmt ebenfalls an, dass in dieser Tabelle eine Spalte namens `id` als Primärschlüssel (*Primary Key*) definiert ist, die via „Auto Increment“ pro Datensatz von dem Datenbankmanagementsystem automatisch hochgezählt wird. Diese beiden Konventionen können jedoch überschrieben werden, sollte das nötig sein.

Wir werden die Klasse `Album` in unserem Model-Verzeichnis ablegen:

zf-tutorial/application/models/Album.php

```
<?php

class Album extends Zend_Db_Table
{

}

}
```

Wie, das war's schon? Richtig! Zum Glück haben wir nur sehr geringe Ansprüche an unserer Anwendung und `Zend_Db_Table` stellt bereits genug Funktionalität bereit, damit wir unsere Ziele erreichen können. Falls du jedoch später spezifische Funktionalität brauchst, um dein Model zu verwalten, gehört sie in diese Klasse. Meistens werden das weitere „find“-Methoden sein, um komplexere Suchvorgänge nach bestimmten Datensätzen umzusetzen.

Alben auflisten

Nun da wir die Konfiguration und den Datenbankzugang aufgesetzt haben, können wir uns nun endlich ans Eingemachte begeben und Alben anzeigen. Das tun wir im Index-Controller.

In jeder Action in `IndexController` wird in irgendeiner Weise die Datenbank befragt. Daher macht es Sinn, die Klasse `Album` bereits im Konstruktor zu laden.

zf-tutorial/application/controllers/IndexController.php

```
<?php

class IndexController extends Zend_Controller_Action
{
    function __construct()
    {
        parent::__construct();
        Zend::loadClass('Album');
    }

    function IndexAction()
    {
        ...
    }
}
```

Dies ist ein Beispiel, für die Benutzung von `Zend::loadClass()` um eigene Klassen zu laden. Es funktioniert, weil wir das Model-Verzeichnis im Bootstrapper (`index.php`) dem PHP include-Pfad hinzugefügt haben.

Wir zeigen alle Alben auf der Index-Action:

zf-tutorial/application/controllers/IndexController.php

```
...
function indexAction()
{
    $view = Zend::registry('view');
    $view->title = "My Albums";

    $album = new Album();
    $view->albums = $album->fetchAll();

    $view->actionTemplate = 'indexIndex.tpl.php';
    echo $view->render('site.tpl.php');
}
...
```

Die Funktion `Zend_Db_Table->fetchAll()` gibt ein `Zend_Db_Table_Rowset` zurück, welches uns erlaubt, später im Template über alle Zeilen in der Tabelle zu iterieren.

zf-tutorial/application/views/indexIndex.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
<p><a href="/zf-tutorial/index/add">Add new album</a></p>
<table>
<tr>
<th>Title</th>
<th>Artist</th>
<th>&nbsp;</th>
</tr>

<?php foreach($this->albums as $album) : ?>
<tr>
<td><?php echo $this->escape($album->title);?></td>
<td><?php echo $this->escape($album->artist);?></td>
<td>
<a href="/zf-tutorial/index/edit/id/<?php echo $album->id;?>"
>Edit</a>
<a href="/zf-tutorial/index/delete/id/<?php echo $album->id;?>"
>Delete</a>
</td>
</tr>
<?php endforeach; ?>
</table>
```

<http://localhost/zf-tutorial/> sollte nun eine schöne Liste mit 2 Alben anzeigen.

Variablen aus POST und GET richtig benutzen

In einer „traditionellen“ PHP-Anwendung werden die „magischen“, globalen Variablen `$_POST` und `$_GET` dazu benutzt, um von Benutzer gesendete Daten abzufragen. Das Problem ist, dass dabei schnell mal die Validierung der Daten vergessen wird, ob sie überhaupt dem erwarteten Typ entsprechen. Wenn keine Validierung vorgenommen wird, treten sehr schnell verschiedene Arten von Sicherheitslücken auf, oder die Anwendung läuft gar nicht mehr. Das Zend Framework stellt deshalb `Zend_Input_Filter` bereit, eine Klasse um die Validierung von Benutzerdaten Daten einfacher zu machen.

Benutzung:

```
$postArray = new Zend_Filter_Input($_POST);
$username = $postArray->testName('username');
if ($username !== false) {
    // $username is a valid name
}
```

Eine Eigenschaft dieser Klasse, die man nicht vergessen sollte, ist, dass es die Originalvariable, die man ihr übergibt, löscht. Nach der Zeile 1 unseres kurzen Beispiels oben, ist `$_POST` null.

Wir richten den Filter wieder gleich im Bootstrapper ein und speichern ihn in der Registry, damit er überall in der Anwendung verfügbar ist. Würden wir ihn nicht in der Registry registrieren, wäre er nur im Bootstrapper verfügbar. Andere, darauf aufbauende Skripte, könnten ihn zwar neu erzeugen, jedoch wäre dann `$_POST` bereits leer.

Betreffender Ausschnitt von zf-tutorial/index.php

```
...
Zend::loadClass('Zend_Db');
Zend::loadClass('Zend_Db_Table');
Zend::loadClass('Zend_Filter_Input');

// register the input filters
Zend::register('post', new Zend_Filter_Input($_POST));
Zend::register('get', new Zend_Filter_Input($_GET));

// load configuration
...
```

Wir können nun den Inhalt einer POST-Variable wie folgt erfragen:

```
$post = Zend::registry('post');
$myVar = $post->testAlpha('myVar');
```

Neue Alben hinzufügen

Jetzt, wo wir den Eingabefilter eingerichtet haben, können wir beruhigt die Funktionalität zum Erstellen eines neuen Albums aufsetzen. Dieser Vorgang besteht aus 2 Teilen:

- ein Formular zur Eingabe der Daten
- eine Verarbeitung der eingegebenen Daten (Speicherung in der Datenbank)

Das ganze werden wir in der addAction() verankern:

zf-tutorial/application/views/IndexController.php

```
...
function addAction()
{
    $view = Zend::registry('view');
    $view->title = "Add New Album";

    if (strtolower($_SERVER['REQUEST_METHOD']) == 'post') {
        $post = Zend::registry('post');
        $artist = trim($post->noTags('artist'));
        $title = trim($post->noTags('title'));

        if ($artist != '' && $title != '') {
            $data = array(
                'artist' => $artist,
                'title' => $title
            );
            $album = new Album();
            $album->insert($data);

            $url = '/zf-tutorial/';
            $this->_redirect($url);
            return;
        }
    }

    // set up an "empty" album
    $view->album = new stdClass();
    $view->album->id = '';
    $view->album->artist = '';
    $view->album->title = '';

    // additional view fields required by form
    $view->action = 'add';
    $view->buttonText = 'Add';

    $view->actionTemplate = 'indexAdd.tpl.php';
    echo $view->render('site.tpl.php');
}
...
```

Beachte die Prüfung von `$_SERVER['REQUEST_METHOD']` mit der wir feststellen, ob das Formular abgeschickt wurde. Falls ja, holen wir aus dem POST-Array den Künstler und den Titel unter Verwendung der `noTags()`-Methode, die sicherstellt, dass kein HTML in der Eingabe (mehr) vorhanden ist. Nachdem wir geprüft haben, ob die Eingaben auch nicht leer waren, setzen wir unser Model `Album` ein, um eine neue Zeile in die Datenbanktabelle einzufügen.

Anschließend bereiten wir das View für das Formular vor, welches wir ins Template einsetzen werden. Vorausschauend können wir nämlich schon sagen, dass das (Formular für das) Hinzufügen und

Bearbeiten von Alben sehr ähnlich ist, daher verwenden wir ein gemeinsames Template, das wir von `indexAdd.tpl.php` und `indexEdit.tpl.php` aus aufrufen werden.

Die Templates für das Hinzufügen eines Albums sehen so aus:

zf-tutorial/application/views/indexAdd.tpl.php

```
<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('_indexForm.tpl.php'); ?>
```

zf-tutorial/application/views/_indexForm.tpl.php

```
<form action="/zf-tutorial/index/<?php echo $this->action; ?>" method="post">
<div>
    <label for="artist">Artist</label>
    <input type="text" class="input-large" name="artist"
        value="<?php echo $this->escape(trim($this->album->artist));?>" />
</div>
<div>
    <label for="title">Title</label>
    <input type="text" class="input-large" name="title"
        value="<?php echo $this->escape($this->album->title);?>" />
</div>

<div id="formbutton">
    <input type="hidden" name="id"
        value="<?php echo $this->album->id; ?>" />
    <input type="submit" name="add"
        value="<?php echo $this->escape($this->buttonText); ?>" />
</div>
</form>
```

Das ist ziemlich einfach gehalten. Weil wir das `_indexForm.tpl.php` Template auch für das Editieren eines Albums benutzen werden, verwenden wir `$view->action`, anstatt die URL hart zu coden. Das gleiche machen wir mit dem Anzeigetext für den Submit-Button des Formulars.

Ein Album bearbeiten

Das Bearbeiten eines Albums läuft wie gesagt fast gleich ab:

zf-tutorial/application/views/IndexController.php

```
...
function editAction()
{
    $view = Zend::registry('view');
    $view->title = "Edit Album";
    $album = new Album();

    if (strtolower($_SERVER['REQUEST_METHOD']) == 'post') {
        $post = Zend::registry('post');
        $id = $post->testInt('id');
        $artist = trim($post->noTags('artist'));
        $title = trim($post->noTags('title'));

        if ($id !== false) {
            if ($artist != '' && $title != '') {
                $data = array(
                    'artist' => $artist,
                    'title' => $title,
                );
                $where = 'id = ' . $id;
                $album->update($data, $where);

                $url = '/zf-tutorial/';
                $this->redirect($url);
                return;
            } else {
```

```

        $view->album = $album->find($id);
    }
} else {
    // album id should be $params['id']
    $params = $this->_action->getParams();
    $id = 0;
    if (isset($params['id'])) {
        $id = (int)$params['id'];
    }
    if ($id > 0) {
        $view->album = $album->find($id);
    }
}

// additional view fields required by form
$view->action = 'edit';
$view->buttonText = 'Update';

$view->actionTemplate = 'indexEdit.tpl.php';
echo $view->render('site.tpl.php');
}
...

```

Beachte, wie wir die Variable `id` holen, wenn wir nicht im POST-Modus sind um die Route zurück zum Bootstrapper zu setzen.

Das Template sieht so aus:

zf-tutorial/application/views/indexEdit.tpl.php

```

<h1><?php echo $this->escape($this->title); ?></h1>
<?php echo $this->render('_indexForm.tpl.php'); ?>

```

Refactoring angesagt!

Es sollte dir nicht entgangen sein, dass wir doppelten Code geschaffen haben, in dem wir für fast identische Vorgänge getrennte Views und Controller verwendet haben. Falls später Änderungen notwendig werden, musst du sie an 2 Stellen vornehmen. Es ist also ein Refactoring angebracht.

Das habe ich dir, werter Leser, als Übung offen gelassen...

Ein Album löschen

Um unsere Anwendung abzurunden, sollten wir noch eine Löschfunktion hinzufügen. Wir haben schon einen Löschen-Link neben jedes Album auf der Index-Action gesetzt und der nächste logische Schritt wäre, ein Album zu löschen, wenn auf den Löschen-Link geklickt wurde. Das wäre aber falsch! Erinnerung an die POST-Spezifikation, man sollte nie eine nicht rückgängig-machbare Aktion via GET, sondern stattdessen via POST laufen lassen. Google's letzte Beta des Accelerator brachte diese Konvention unter die Massen.

Wir sollten eine Bestätigungsabfrage bereitstellen, die erscheint, wenn der Benutzer auf „Löschen“ klickt – falls er bejaht, wird der Löschvorgang durchgeführt.

Der Code sieht ähnlich aus wie der, der Hinzufügen- beziehungsweise Bearbeiten-Actions:

zf-tutorial/application/views/IndexController.php

```

...
function deleteAction()
{
    $view = Zend::registry('view');
    $view->title = "Delete Album";
}

```

```

$album = new Album();
if (strtolower($_SERVER['REQUEST_METHOD']) == 'post') {
    $post = Zend::registry('post');
    $id = $post->getInt('id');
    if (strtolower($post->testAlpha('del')) == 'yes' && $id > 0) {
        $where = 'id = ' . $id;
        $album->delete($where);
    }
} else {
    // album id should be $params['id']
    $params = $this->_action->getParams();
    if (isset($params['id'])) {
        $id = (int)$params['id'];
        if ($id > 0) {
            $view->album = $album->find($id);
            $view->actionTemplate = 'indexDelete.tpl.php';
            // only render if we have an id.
            echo $view->render('site.tpl.php');
            return;
        }
    }
}
// redirect back to the album list in all cases unless we are
// rendering the template
$url = '/zf-tutorial/';
$this->_redirect($url);
}
...

```

Wir benutzen wieder denselben Trick, den Inhalt von `$_SERVER['REQUEST_METHOD']` zu prüfen, um herauszufinden, wann wir die Bestätigungsabfrage anzeigen oder das Album, welches wir anhand der ID identifizieren, löschen sollen. Ähnlich wie das Einfügen oder Bearbeiten von Zeilen in eine Tabellen, wird das Löschen letztlich durch den Aufruf von `Zend_Db_Table::delete()` durchgeführt.

Beachte auch das `return`-Statement direkt nach dem Rendern des Templates. Damit können wir nach dem Löschen direkt wieder zur Alben-Übersicht zurückkehren. Falls irgendeine unserer Plausibilitätsprüfungen fehlschlägt, können wir so direkt zum Index zurückkehren, ohne den `_redirect()`-Aufruf mehrere Male schreiben zu müssen.

Das Template ist ein einfaches Formular:

zf-tutorial/application/views/indexDelete.tpl.php

```

<h1><?php echo $this->escape($this->title); ?></h1>
<?php if ($this->album) :?>
<form action="/zf-tutorial/index/delete" method="post">
    <p>Are you sure that you want to delete
        '<?php echo $this->escape($this->album->title); ?>' by
        '<?php echo $this->escape($this->album->artist); ?>'?</p>
    <div>
        <input type="hidden" name="id"
            value="<?php echo $this->album->id; ?>" />
        <input type="submit" name="del" value="Yes" />
        <input type="submit" name="del" value="No" />
    </div>
</form>
<?php else: ?>
<p>Cannot find album.</p>
<?php endif;?>

```

Fazit

Wir schließen damit den Aufbau unserer simplen, jedoch voll funktionsfähigen MVC-Anwendung unter Verwendung des Zend Framework ab. Ich hoffe, ich habe euer Interesse geweckt und konnte euch die Funktionsweise einer Applikation nach dem Model-View-Controller-Muster nahe bringen. Falls ihr Fehler im Tutorial findet, lasst es mich unter rob@akrabat.com wissen. Fehler die Übersetzung betreffend, könnt ihr gerne dem Blog von Daniel Messer unter blog@selfphp.org melden.

Dieses Tutorial hat nur die elementaren Grundlagen des Frameworks betrachtet, es gibt jedoch sehr viele weitere Klassen zu entdecken! Ich kann euch nur empfehlen euch die [Dokumentation](#) und das [Wiki](#) anzusehen um weitere Einblicke zu erhalten. Es lohnt sich in jedem Fall!

Letzte Gedanken

Während der Entwicklung dieses Tutorials war das Auffälligste, was gefehlt hat, ein besserer Weg um Model-Klassen zu implementieren. Ich kann gut nachvollziehen, warum das ActiveRecord-Muster zur Zeit sehr populär ist. Es gibt dafür bereits einen Vorschlag im Wiki des Zend Frameworks für eine `Zend_Db_Model`-Klasse, die wie folgt beschrieben ist:

Ein Objekt, welches einen Wrapper für einen Datensatz in einer View oder Tabelle darstellt, den Zugriff kapselt und es erlauben soll, den Daten Aspekte aus der Fachlogik hinzuzufügen.

Eine Entwicklung in diese Richtung, wäre sicherlich sehr hilfreich, um Anwendungen komplett mit dem Zend Framework zu betreiben.

Alles in Allem, macht das Zend Framework schon jetzt einen sehr guten Eindruck!

Anmerkung des Übersetzers: Mit dem Begriff View ist nicht die Anzeigekomponente aus dem MVC-Framework gemeint, sondern eine [Sicht](#) auf eine oder mehrere Tabellen einer Datenbank. Wem das ActiveRecord-Muster nicht bekannt ist: es beschreibt ein Objekt, dessen Klasse eine Tabelle repräsentiert und deren Eigenschaften die Spalten einer solchen. Jede Instanz eines solchen Objekts repräsentiert genau einen Datensatz aus der Tabelle.