Eine Business Applikation mit dem Zend Framework

Die Entwicklung von Business-Applikationen kann schnell beängstigende Größen annehmen. Das Zend Framework [1] als Basis erlaubt die Konzentration auf das Wesentliche und beschleunigt die Entwicklung. Wir wollen hier am Beispiel einer Mini-Applikation und auf Grund unserer eigenen Erfahrungen mit dem Zend Framework den Einstieg erleichtern.

In diesem Artikel erfahren Sie...

- Einstieg ins Zend Framework anhand einer Mini-Applikation. Insbesondere wird exemplarisch der Aufbau der notwendigen Bootstrap-Datei erläutert.

Was Sie vorher wissen sollten...

- Grundkenntnisse in PHP und der objektorientieren Programmierung;
- Grundkenntnisse in der Konfiguration des Apache Webservers, um die Beispiele nachzuvollziehen.

Schwierigkeitsgrad:

Am Anfang der Entwicklung unserer Applikation stand die Überlegung, welche Voraussetzungen eine Applikation erfüllen muss, die sich primär an Kunden im Bereich mittlerer und großer Unternehmen wendet. Ganz oben auf der Wunschliste stand dabei die Datenbankabstraktion, d.h. die Anwendung sollte sowohl unter der von uns für die Entwicklung benutzten MySQL Datenbank laufen, als auch unter PostgreSQL, MS-SQL, Oracle und DB2.

Gleich als nächstes hatten wir dann die Anforderung der Mehrsprachigkeit. So musste gewährleistet sein, dass nach dem Login alle Bezeichner in der jeweils gewählten Sprache zur Verfügung stehen. Für die Entwicklung haben wir uns hier auf deutsch und englisch beschränkt, das System aber so aufgebaut, dass es *n* Sprachen unterstützt.

Die Anmeldung bzw. Authentifizierung des Users am System, sollte sowohl über die Userverwaltung der Applikation als auch über LDAP möglich sein. Die Benutzerverwaltung in größeren Unternehmen erfolgt in der Regel über LDAP, bzw. Active Directoty.

Das Applikationsdesign

Zu Beginn unserer Entwicklung war das Zend Framework erst in der Version 0.8 verfügbar. Wir haben uns trotzdem dafür entschieden, da wir der Auffassung sind, dass es gerade für große Web-Applikationen eine hervorragende Basis darstellt. Wichtig für die Entscheidung waren auch die Dokumentation [2] und die einfache Einrichtung des Frameworks.

Die Dokumentation steht neben englisch und deutsch noch in vielen weiteren Sprachen zur Verfügung und ist sehr umfangreich.

Die Einrichtung des Zend Frameworks ist im Gegensatz zu anderen Frameworks sehr einfach. Die Generierung von Konfigurationsdateien oder SQL Befehlen ist nicht notwendig.

MVC Design Pattern

Das Framework implementiert das Model-View-Controller Entwurfsmuster (engl.: MVC Design Pattern). Durch den Einsatz des MVC-Patterns ist es möglich, Anwendungs- und Anzeigelogik sauber voneinander getrennt zu halten. Dabei übernimmt die Model-Schicht den Umgang mit den Daten – hier wird im Prinzip die Kommunikation mit der Datenbank abgewickelt. Die View-Schicht kümmert sich dagegen um die Präsentation der Daten – in der Regel also um das Rendering der Templates. Die eigentliche Steuerung der Programmlogik findet dann im jeweiligen Controller statt. Sehr zu empfehlen zum ersten Kontakt mit dem Framework ist [2].

Verzeichnisstruktur

Die von uns verwendete Verzeichnisstruktur ist eine mögliche Herangehensweise und ist in Listing 1 zu finden. Die eigentlichen Controller-, Model- und View-Skripten liegen dabei in den jeweiligen Unterverzeichnissen von `application`.

Das `htdocs` Verzeichnis beinhaltet alle öffentlich erreichbaren Dateien, insbesondere unsere Bootstrap-Datei index.php (...).

Im `library` Verzeichnis werden alle Dateien des Zend Frameworks unter `Zend` abgelegt. Im Verzeichnis `ext` unterhalb von `library` werden Klassen abgelegt, die das Framework erweitern. Zum einen ist hier der richtige Ort für eigene Klassen, die global in der Applikation verfügbar sein sollen. Zum anderen aber auch so genannte Adapter, die Klassen des Zend Frameworks überladen können. Ein Beispiel für einen Adapter ist z.B. eine Klasse, die `Zend_DB` überlädt.

Webservereinrichtung – VirtualHost

Damit nun auch alle Anfragen tatsächlich über unsere Bootstrap-Datei (s. nächster Abschnitt) laufen, muss im Webserver (in unserer Applikation Apache 2) die Rewrite-Engine angeschaltet werden. Einen exemplarischen VirutalHost Conatiner zeigt Listing 2. Während der Entwicklung kann man hier den Rewrite LogLevel ruhig auf 6 setzen. Damit lässt sich dann sehr gut nachvollziehen, ob auch alles wie gewünscht funktioniert.

Die Rewrite-Condition in Zeile 9 besagt, dass alle URIs die nicht existieren (!-f) von der Bootstrap-Datei `index.php` beantwortet werden. Anfragen für existierende Dateien, z.B. Bilder aus dem `/htdocs/img` – Verzeichnis werden hingegen durch den Webserver direkt ausgeliefert.

Das zentrale Element die Bootstrap-Datei index.php

Nun zum Aufbau der Bootstrap-Datei (`index.php`) im Listing 3.

Damit alle Dateien des Frameworks wie gewünscht geladen werden können, müssen wir die zusätzlich benötigten Pfade noch in den include_path der Anwendung durch die PHP Funktion `set_include_path()` mit aufnehmen (Zeile 12-17).

Um das automatische Laden der Klassen kümmert sich dann der `Zend_Loader`, so dass

wir keine weiteren Anweisungen der Form `require_once` mehr benötigen (Zeile 19-22). Das macht Sinn, da die Klassennamen vollkommen dynamisch übergeben werden. In der Praxis muss nur unterhalb des Verzeichnisses `application/controllers` ein neuer Controller erstellt werden. Dieser wird automatisch durch Zend_Loader in das System eingebunden. Natürlich darf dabei ggf. ein Model in `application/models` und ein View in `application/views` nicht vergessen werden.

Zeile 25 zeigt einen weiteren Anwendungsfall für Zend_Loader: Wir können einfach per:

```
$db = new DbConnector();
```

ein Objekt der DB Klasse zur Verfügung stellen.

Gleich danach verwenden wir eine weitere sehr komfortable Funktion des Frameworks: `Zend_Registry`. Wir verwenden hier z.B.:

```
Zend_Registry::set('db', $db);
```

um Objekte oder Werte innerhalb der Anwendung global verfügbar zu machen. Innerhalb eines Controllers, Models oder einer eigenen Klasse kann dann mit:

```
$this->db = Zend_Registry::get('db');
```

einfach auf das Objekt zugegriffen werden.

Die Mini-Applikation

Wir haben für das einfachere Verständnis und zu Anschauungszwecken eine Mini-Applikation für Sie erstellt. Auf der Heft CD oder auf unserer Website [3] finden Sie den kompletten Code.

Es handelt sich bei der Mini-Applikation um einen virtuellen mikro-shop mit einem Login, einem Logout, einer Kunden- und einer Produktübersicht. Anhand der Mini-Applikation können Sie alles in diesem Artikel besprochene nachvollziehen oder diese Applikation als Grundlage für Ihr eigenes Projekt verwenden.

Wichtige Zend Framework Elemente im Detail

Im Folgenden betrachten wir nun eine Auswahl von wichtigen Elementen des Frameworks etwas genauer. Es sind die Elemente, die auch in der Mini-Applikation eingesetzt werden.

Zend_Auth

Da die User sich vor Benutzung des Systems natürlich anmelden sollen, müssen wir eine Möglichkeit der Authentifizierung schaffen. Wir erstellen hier in Zeile 29 der Bootstrap-Datei zunächst ein neues Zend_Session_Namespace Objekt durch:

```
$authSession = new ZendSession_
   Namespace('auth');
```

Dieses ist im Prinzip die Zend-Variante des normalen PHP Session-Handlings. Damit die Session später in den notwendigen Controller- und Model-Dateien zur Verfügung steht, registrieren wir sie in Zeile 30 mit:

```
Zend_Registry::set('authSession',
   $authSession);
```

Als nächstes erstellen wir in Zeile 33 ein Objekt von Zend_Auth und machen das Objekt durch den Zend_Registry Mechanismus global verfügbar.

In den Zeilen 36-48 werden dann die entsprechenden Adapter aufgerufen, um zu prüfen, ob die Userdaten gültig sind oder nicht. War der Login erfolgreich, findet sich in:

```
$authSession->userId;
```

dann die Identität des Users wieder.

Damit das auch alles tatsächlich so funktioniert muss per:

```
$ctrl->registerPlugin(new PluginAuth
   ($auth));
```

in Zeile 68 das Plugin für die User-Authentifizierung zunächst einmal in das Controller-Objekt *geklemmt* werden. Diese Plugin-Technik hat den Vorteil, dass man hier an den so genannten *preDispatch*-Prozess gelangt. Das bedeutet, dass der Controller jeweils vor Verteilung der eigentlichen Aufgaben immer erst einmal prüft, ob es sich tatsächlich um einen gültigen Benutzer handelt. Sollte dies nicht der Fall sein, wird schlicht der Controller ausgetauscht und ein nicht angemeldeter User wird immer nur den Login-Screen zu sehen bekommen. So soll's ja auch sein.

Damit ist das Thema Authentifizierung erst einmal erledigt und wir können uns um die eigentliche Applikation kümmern.

In der Mini-Applikation verwenden wir eine lokale Datenbank, um die Userdaten zu speichern. An dieser Stelle könnte man aber auch ohne weiteres eine Authentifizierung gegen einen LDAP-Server vornehmen. Zwar bietet das Zend Framework hier bislang noch keine speziellen Adapter an, grundsätzlich spricht aber nichts dagegen hier per PHP Funktion `ldap_connect()` einen LDAP-Server als Userdatenbank zu verwenden.

Routing

Durch das hier verwendete Routing (Zeile 51-54) wird eine URL nach einem bestimmten Schema (Zeile 51) zerlegt und die Anfrage an den zuständigen Controller und dessen entsprechende Methode weitergeleitet. Beide Werte sind hier jeweils mit `index` vorbelegt. Wird also nichts weiter in der URL übergeben (wenn z.B. `/` aufgerufen wird), wird der IndexController mit seiner Methode `IndexAction()` angesprungen. In unserer Mini-Applikation sind die URLs nach folgendem Schema aufgebaut:

```
http://localhost/[Controller]/[Action]/
   [Key1]/[Value1]/[Key2]/[Value2]
```

damit würde dann:

http://localhost/products/edit/id/384

den Controller `ProductsController.php` und in diesem die Methode `editAction()` aufrufen. Alle nachfolgenden Elemente der URI sind dann nach dem Prinzip key/value aufgebaut und stehen innerhalb des Controllers als Parameter zur Verfügung.

Zend_Config

Das Controller- und das View-Objekt brauchen Pfadangaben, um zu ermitteln, wo die notwendigen Skripte liegen. Wir definieren diese Pfade erst einmal ganz einfach in einem Array (Zeile 57-59) und machen daraus dann ein `Zend_Config` Objekt (Zeile 61). Zend_Config erleichtert den Umgang mit Konfigurationsdaten, die entweder wie hier als Array, oder als XML- bzw.

Listing 1. *Verzeichnisstruktur*

```
1  /application
2     /controllers
3     /models
4     /views
5  /htdocs
6     /css
7     /img
8     index.php
9  /library
10    /ext
11    /Zend
```

Listing 2. *Beispiel VirtualHost Container*

```
1  <VirtualHost 127.0.0.1>
2  ServerName projekt1
3  <Directory /var/www/projekt1/
     htdocs>
4  Options All
5  </Directory>
6  DirectoryIndex index.php
7  DocumentRoot /var/www/projekt1/
     htdocs
8  RewriteEngine On
9  RewriteCond %{DOCUMENT_ROOT}/
     %{REQUEST_URI} !-f
10  RewriteRule .* /index.php [L]
11  RewriteLog  /var/log/apache2/
     projekt1_rewrite.log
12  RewriteLogLevel 6
13  php_flag asp_tags on
14  php_flag display_errors on
15  php_value short_open_tag off
16  </VirtualHost>
```

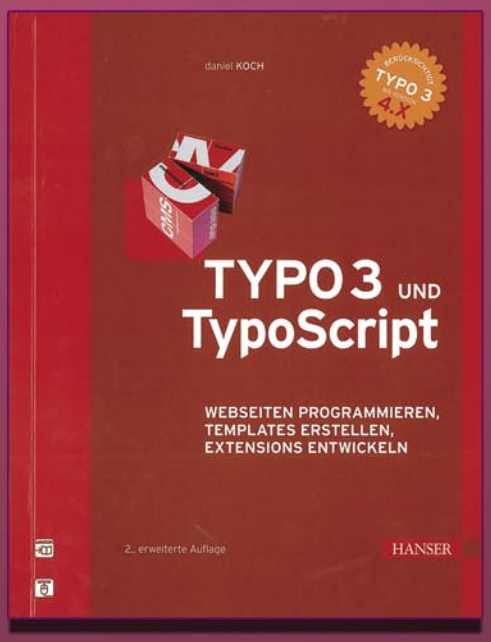
Listing 3. *Die Bootstrap Datei*

```
<?php
error_reporting(E_ALL ^ E_NOTICE);
define('DOCROOT', $_SERVER['DOCUMENT_ROOT']);
define('APPROOT', preg_replace('|/[^/]+/?$|', '',
                  DOCROOT));

define('DB_HOST', 'localhost');
define('DB_NAME', 'projekt1');
define('DB_USER', 'projekt1_user');
define('DB_PASS', 'projekt1_pass');
 define('DB_DRIVER', 'PDO_Mysql');

// Include Paths
$includePath  = get_include_path();
$includePath .= PATH_SEPARATOR . APPROOT . "/library/";
$includePath .= PATH_SEPARATOR . APPROOT . '/library/
                  ext';
$includePath .= PATH_SEPARATOR . APPROOT . '/application/
                  models';
set_include_path($includePath);

require_once 'Zend/Loader.php';
function __autoload($class){
   Zend_Loader::loadClass($class);
}

// Database
$db = new DbConnector();
Zend_Registry::set('db', $db);

// Session
$authSession = new Zend_Session_Namespace('auth');
Zend_Registry::set('authSession', $authSession);

// Authentication
$auth = Zend_Auth::getInstance();
Zend_Registry::set('auth', $auth);

if (!$auth->hasIdentity()) {
   if (!empty($_POST['__login__'])) {
      $authAdapter = new Authentication($_
                  POST['username'], $_POST['password']);
      $authResult = $auth->authenticate($authAdapter);

      if ($authResult->isValid()) {
         $authSession->userId = $auth->getIdentity();
         $authArr = $authResult->getMessages();
      }

      $authSession->authMessage = $authResult-
                  >getMessages();
   }
   }

// Routing
   $route = new Zend_Controller_Router_Route(':
                  controller/:action/*',
                  array('controller' => 'index', 'action'
                  => 'index'));

$router = new Zend_Controller_Router_Rewrite();
$router->addRoute('actionroute', $route);
Zend_Registry::set('router', $router);

// Set the Paths
$conf['path']['view']      = APPROOT . '/application/
                  views/';

$conf['path']['ctrl']      = APPROOT . '/application/
                  controllers';

$conf['path']['helper']      = APPROOT . '/library/ext/
                  viewHelper/';


// Configuration
$config = new Zend_Config($conf);
Zend_Registry::set('config', $config);

// Front-Controller
$ctrl = Zend_Controller_Front::getInstance();
$ctrl->setRouter($router);
$ctrl->registerPlugin(new PluginAuth($auth));
$ctrl->setControllerDirectory(array($config->path->ctrl));
$ctrl->throwExceptions(true);
$ctrl->returnResponse(false);
$ctrl->setParam('noViewRenderer', true);

// Loading the view
$view = new Zend_View(array('encoding' => 'UTF-8'));
$view->setScriptPath($config->path->view);
Zend_Registry::set('view', $view);
$view->setHelperPath($config->path->helper, 'My_View_
                  Helper');


// Dispatch process
try {
   $ctrl->dispatch();
   $response = $ctrl->getResponse();
}
catch (Exception $e) {
   require_once($config->path->ctrl . '/
                  IndexController.php');

   IndexController::norouteAction();
   echo $e->getMessage();
}

$info = $ctrl->getRequest()->getParams();

$view->assign('topnav', 'top.php');
$view->assign('footer', 'footer.php');

$view->data = array(
      'title'         => 'Test Applikation',
      'info'         => $info,
      );

 echo $view->render('main.php');

 ?>
```

phpsolutions

Neue Technologien und Lösungen für PHP-Programmierer

prämien* – abo



Für alle speziell Willkomensgeschenk:

Archivausgaben von PHP Solutions

abo@software.com.pl

Bestellformular auf S. 71

*zur Wahl

INI-Datei vorliegen. Später kann dann auf die Werte der Konfiguration mit der Syntax:

```
$config->[Name der Sektion]->[Variablenname];
```

zugegriffen werden, in unserem Beispeil also per:

```
$config->path->view;
```

Zend_View

Nachdem wir dann noch den Controller mit dem oben erwähnten Plugin für die Authentifizierung erstellt haben (Zeile 66-72), erstellen wir das View Objekt in Zeile 75. Dies ist, wenn man so will, die Darstellungsschicht der Applikation.

Es spricht nichts dagegen ein externes Template System wie Smarty oder FastTemplate zu nutzen. Wir sind jedoch davon abgekommen, da ein Template System für eine hoch dynamische Applikation immer eine Performance Einbuße mit sich bringt. `Zend_View` beinhaltet standardmäßig so genannte Helper für alle Formularelemente. Weiterhin ist es möglich so genannte View_Helper selbst zu erstellen und einzubinden. Dazu später mehr. Mit diesen Tools und mit if ... then ... else und foreach Schleifen, lassen sich auf einfache Weise alle notwendigen Anforderungen in einem View umsetzen. Schließlich ist PHP ja ursprünglich als Template Sprache von Rasmus Lerdorf entwickelt worden.

Dispatch Prozess

Die Verteilung der Anfrage an den richtigen Controller und dessen Methode wird dabei über den Dispatch-Prozess abgewickelt (Zeile 81-89). Läuft hier etwas schief, weil z.B. eine fehlerhafte URL und somit ein nicht existierender Controller und/oder Methode aufgerufen wird, greift der catch-Abschnitt und leitet die Anfrage auf den Index-Controller und dessen Methode `norouteAction()`.

In Zeile 91 werden dann noch einmal alle ggf. übergebenen Parameter zur weiteren Verwendung in ein assoziatives Array gepackt. Das Framework übernimmt hier netterweise nicht nur die Parameter, die wir in der URL in der Form `.../key1/value1/...` notiert haben, sondern auch alle weiteren Parameter die per GET und/oder POST gesendet wurden. Innerhalb der Controller machen wir diese Parameter im Constructor jeweils mit:

```
$this->params = $this->getRequest()
   ->getParams();
```

verfügbar.

View Skripte

Es fehlt noch die Zuweisung der einzelnen View-Dateien. Für die zumeist mit wenig Dynamik behafteten Kopf und Fußzeilen des Projekts verwenden wir hier in der Index-Datei:

```
$view->assign('topnav', 'top.php');
$view->assign('footer', 'footer.php');
```

Wir weisen hier der Array-Variablen $view->data noch ein paar Daten zu. Innerhalb des Haupt-Views `main.php` können wir so z.B. auf den Titel der Applikation einfach per:

```
$this->data['title'];
```

zugreifen. Die letztendliche Ausgabe des finalen Contents findet dann in Zeile 101 per:

```
echo $view->render('main.php');
```

statt.

Ein Beispiel – die Mini-Applikation

Unsere Mini-Applikation soll es ermöglichen, nach erfolgreichen Login jeweils eine Liste von Produkten und Kunden darzustellen und die einzelnen Einträge zu bearbeiten. Dazu benötigen wir für die einzelnen Bereiche mindestens ein Controller-Skript und ein View-Skript. Dabei wird der jeweilige View dynamisch im entsprechenden Controller geladen. In Listing 4 sehen Sie den notwendigen Controller mit seinen Methoden `indexAction()` als Einstiegsmethode, die eine Übersichtsliste vorhandener Produkte anzeigt. Außerdem die

Listing 4. *Ein Beispiel-Controller*

```
1  <?php
2
3  class ProductsController extends Zend_Controller_Action {
4
5     public function __construct(Zend_Controller_Request_Abstract $request,
6        Zend_Controller_Response_Abstract $response, $invokeArgs = array()) {
7        parent::__construct($request, $response, $invokeArgs);
8        $this->view = Zend_Registry::get('view');
9
10        $this->prod = new ProductsModel();
11
12        $this->params = $this->getRequest()->getParams();
13
14     }
15
16     public function indexAction() {
17
18        if ($this->params['__save__']) {
19           $this->prod->update($this->params);
20        }
21
22        $this->view->headline = 'Produkt-&Uuml;bersicht';
23
24        $this->view->products = $this->prod->getProducts();
25
26        $this->view->assign('content', 'products/index.php');
27
28     }
29
30     public function editAction() {
31
32        $product = $this->prod->getDetails($this->params['id']);
33
34        $this->view->headline = 'Produkt bearbeiten';
35        $this->view->title = $this->view->formText('title', $product['title'],
              array('size' => 50));
36        $this->view->price = $this->view->formText('price', $product['price'],
              array('size' => 10));
37        $this->view->id = $this->view->formHidden('id', $product['id']);
38        $this->view->submit = $this->view->formSubmit('__save__', 'Speichern');
39
40        $this->view->assign('content', 'products/edit.php');
41
42     }
43  ...
44  }
45  ?>
```

Methode `editAction()`, die es ermöglicht einen Eintrag zu bearbeiten.

Zend_View Helper

Bei der Darstellung der Standardelemente für HTML-Formulare greift einem das Zend Framework unter die Arme. Hier kann z.B. mit Hilfe des vorhandenen `View_Helpers formText` ein Input-Element vom Typ Test erzeugt werden:

```
$this->view->title = $this->view
   ->formText('title', $product['title'],
    array('size' => 50));
```

Der erste Parameter gibt hierbei den Namen des zu erzeugenden Elements an, der zweite den Wert. Weitere Attribute des Elements können dann als dritter Parameter in einem Array mitgegeben werden. Hier würde also das Input-Element wie folgt erzeugt werden:

```
<input type="text" name="title" id="title"
   value="Produkt 1" size="50" />
```

Variablen für das View-Skript, welches im Falle der `indexAction()` Methode per:

```
$this->view->assign('content',
   'products/index.php');
```

eingebunden wird, können im Controller einfach per:

```
$this->view->headline = 'Eine Überschrift';
```

zugewiesen werden. Da das View-Skript innerhalb der Instanz des Zend_View ausgeführt wird, steht diese Variable dort dann per:

```
$this->headline;
```

zur Verfügung. Wir haben hier die View-Skripte jeweils in ein Verzeichnis pro Controller untergliedert und in der Regel pro Methode ein Skript angelegt. Das hat den Vorteil, dass man sich auch bei großen Applikationen mit vielen Views schnell zurecht findet.

Listing 5. *Die eigene Helper Klasse*

```
1   <?php
2
3   class My_View_Helper_JsTooltip {
4
5    public function JsTooltip($id,
                        $text) {
6
7     // irgend welche Logik
8
9        $out  = $text;
10       $out .= "hier steht Content";
11
12    return $out;
13    }
14   }
15  ?>
```

Eigene Helfer

Das Zend Framework stellt wie schon oben beschrieben standardmäßig Helper zur Verfügung, mit denen alle HTML Formularelemente abgebildet werden können. Außerdem ist es möglich eigene Helper zu erzeugen und diese in das Framework zu integrieren. Dabei sollte man darauf achten, dass die eigenen Helper Klassen außerhalb des Zend Verzeichnisses liegen (bei uns library/Zend). Somit wird gewährleistet, dass bei einem update des Frameworks die eigens erstellten Helper nicht überschrieben werden.

In unserer Mini-Applikation nutzen wir keine Helper. Die grundlegende Funktionsweise und Einbindung soll aber trotzdem kurz beschrieben werden. Zu aller erst müssen wir in in unserer Bootstrap-Datei den/die Pfad/e zu den eigenen Helper Klassen mitteilen. Ein sinnvoller Ort ist z.B.: `library/ext/viewHelper`. Das setzen des Pfades erfolgt durch:

```
$view->setHelperPath('/pfad/zu/den/
   Helperklassen/', 'My_View_Helper');
```

Das erste Argument ist der absolute Pfad zu den Helperklassen. Das zweite Argument ist ein eigener Präfix, um einen eigenen Namensraum zu bestimmen. Eine Klasse JsTooltipp muss dann

```
My _ View _ Helper _ JsTooltipp
```

heißen. Durch die Nutzung der Methode `addHelperPath()` ist es nun möglich noch weitere Pfade anzugeben, um einen Stapel zu errichten. `Zend_View` geht alle Pfade durch und sucht nach dem angeforderten Helper:

```
$view->addHelperPath('/pfad/zu/den/Weitern/
   Helperklassen/', 'Other_View_Helper');
```

Prinzipiell erstellt man für jeden eigenen Helper eine Klasse. Den Aufbau der Klasse entnehmen Sie bitte Listing 5. Der Aufruf des Helper's innerhalb eines Controllers, Models oder Views erfolgt durch:

```
$view->JsTooltip('1', 'hallo');
```

Die Erstellung eigener Helper Klassen ermöglicht uns `Zend _ View` so zu erweitern, dass wir ein ausgereiftes Template System nach unseren eigenen Anforderungen erstellen können. Prinzipiell ist jede Art Logik möglich und umsetzbar. Die Ersparnis an Tipparbeit ist immens und wir nutzen die Objektorientierung weiterhin voll aus.

Soviel zu der prinzipiellen Nutzung von eigenen Helperklassen mit `Zend_View`. Die Dokumentation unter [4] dient dem tieferen Einblick.

Zend_Db

Um jetzt zunächst einmal eine Liste der Produkte anzeigen zu können, bedienen wir uns der dazugehörigen Model-Datei (Listing 6), die im Konstruktor der Controllers per:

```
$this->prod = new ProductsModel();
```

eingebunden wurde. Hier würde also:

```
$this->view->products = $this->prod
   ->getProducts();
```

alle verfügbaren Produkte in ein Array laden.

Angenehm fällt hierbei die Arbeit mit `Zend_Db_Select` auf. `Zend_Db_Select` übernimmt die Abstraktion der SELECT Statements für die verwendete Datenbank. Um das Risiko von SQL Injections zu minimieren wird außerdem das korrekte Quoting von Bezeichnern und Werten vorgenommen. Zu beachten ist hierbei, dass vor der Verwendung des Datenbank-Objekts mit:

```
$this->db = Zend_Registry::get('db');
$this->selObj = $this->db->dbObj->select();
```

dieses immer erst einmal mit:

```
$this->selObj->reset();
```

geleert wird. Andernfalls kann es bei mehrfacher Verwendung des SELECT Objekts dazu kommen, dass noch Reste alter Statements vorhanden sind und ggf. ausgeführt werden. Ein:

```
"select id, title, price from products
   order by title limit 0, 10";
```

im MySQL-Jargon würde hier geschrieben werden als:

```
$this->selObj->reset();
$this->selObj->from('products',
   array('id', 'title', 'price'));
$this->selObj->order('title');
$this->selObj->limit(0,10);
$data = $this->db->dbObj
   ->fetchAll($this->selObj);
```

Spätestens bei einem Wechsel der Datenbank weiß man solche Konstrukte zu schätzen, da man sich um so MySQL-spezifische Dinge wie *limit* nicht mehr kümmern muss.

Will man jetzt ein gegebenes Produkt editieren, wird durch Aufruf der URL `/products/edit/id/4` automatisch die Methode `editAction()` im Products-Controller mit

den Parametern `id => 4` aufgerufen. Die benötigten Detailinformationen über das Produkt liefert auch hier wieder das Model per:

```
$product = $this->prod->getDetails
   ($this->params['id']);
```

Danach enthält die Variable $product alle benötigten Informationen.

Daten speichern

Die Action unseres Edit-Formulars zeigt `auf /products/index`. Werden Daten gesendet, wird die Methode `update()` des Products-Models aufgerufen.

Da es sich in diesem Beispiel um eine sehr einfache Struktur handelt, könnte man in diesem Fall natürlich auch die `update()` Methode von `Zend_Db_Table` verwenden. In größeren Projekten ist es aber oft so, dass es bei einer Datensatzaktualisierung nicht mit einem Statement getan ist, sondern ggf. mehrere Tabellen aktualisiert werden müssen. Aus diesem Grund haben wir zu Beginn der Entwicklung festgelegt, dass alle schreibenden Operationen auf der Datenbank in Transaktionen gekapselt werden müssen. Dazu muss man sich allerdings vom MyISAM Tabellenformat der MySQL verabschieden.

Deshalb wählten wir als Storage Engine InnoDB. InnoDB ist ab Version 5 des MySQL-Servers standardmäßig enthalten. Mit Hilfe des `Zend_Db_Adapters` kann man Transaktionen innerhalb der Applikation relativ leicht gewährleisten. Wir haben hierzu eine zentrale Klasse definiert (Listing 7), so dass unsere Transaktionen dann in etwa so aussehen:

```
$qs = "delete from sometable where id = 5";
$this->db->sqlAdd($qs);
$qs = "insert into sometable (id, title)
   values (1, 'test');
$this->db->sqlAdd($qs);
$this->db->transaction();
```

Damit werden die Statements entweder ganz oder gar nicht ausgeführt. Waren alle Statements erfolgreich, wird ein COMMIT ausgeführt und die Änderung an der Datenbank sind dauerhaft. Tritt jedoch ein Fehler an einer beliebigen Stelle auf, wird ein ROLLBACK ausgelöst und die Datenbank kommt wieder in ihren ursprünglichen Zustand. Wir laufen keine Gefahr mehr die Daten in einen inkonsistenten Zustand zu bringen.

Listing 6. *Ein Beispiel Model*

```
1   <?php
2
3   class ProductsModel {
4
5      public function __construct() {
6
7         $this->db = Zend_Registry::get('db');
8         $this->selObj = $this->db->dbObj->select();
9
10        }
11
12     public function getProducts() {
13
14        $this->selObj->reset();
15        $this->selObj->from('products',
              array('id', 'title', 'price'));
16        $this->selObj->order('id');
17        $data = $this->db->dbObj->fetchAll($this->selObj);
18
19        return $data;
20
21        }
22
23     public function getDetails($id) {
24
25        $this->selObj->reset();
26        $this->selObj->from('products');
27        $this->selObj->where('id = ?', $id);
28        $data = $this->db->dbObj->fetchAll($this->selObj);
29
30        return $data[0];
31
32        }
33
34     public function update ($data) {
35
36        $qs = sprintf("update products set title = '%s',
              price = '%s' where id = %s",
             $data['title'], $data['price'], $data['id']);
37        $this->db->sqlAdd($qs);
38        $this->db->transaction();
39
40
41     }
42
43  }
44
45  ?>
```

Tschüss MySQL

Trotz der Eleganz der Transaktionen, stießen wir dann doch an die Grenzen des Datenbanksystems MySQL.

Die Herausforderung war, an Inhalten der Datenbank eine Volltextindizierung vorzunehmen. Leider ist dies bei MySQL nur für Tabellen möglich, die das Tabellenformat MyISAM verwenden. Da wir uns jedoch bereits auf Grund der Transaktionssicherheit für das Tabellenformat InnoDB entschieden hatten, sahen wir uns hier vor die Wahl gestellt. Wir entschieden uns dann für einen kurzfristigen Bruch mit der MySQL und portierten unser System in der Basiskonfiguration kurzerhand auf die PostgreSQL Datenbank [5]. Dank der verwendeten Abstraktionsschicht Zend_Db war die Umstellung inkl. der Grundeinrichtung des PostgreSQL Clusters innerhalb eines Tages erledigt.

Was man natürlich im ersten Moment vermisst, ist ein so komfortables Administrationstool wie das web basierte phpMyAdmin[6]. Zwar gibt es auch hier mit phpPgAdmin [7] etwas ähnliches, wir entschieden uns jedoch für die Client-Anwendung pgAdmin [8]. Dieses Tool lässt von der Administration her keine Wünsche offen.

Arbeitet man mit einer lokalen Datenbank, kann man dieses natürlich sofort verwenden. Spätestens im produktiven Betrieb wird die Datenbank aber auf einem entfernten Datenbank- / Webserver liegen. Um jetzt keine unnötigen Sicherheitslöcher zu erzeugen (indem man beispielsweise den Port 5432 des Servers, auf dem die Postgres lauscht, öffnet), schickt man die Verbindung von PgAdmin zum Server am besten

durch einen SSH Tunnel. Diesen erzeugt man bei einem Linux System auf der Client-Seite mit:

```
ssh user@remotehost -L 5432:localhost:5432
```

Damit werden dann alle Anfragen auf den Port auf der lokalen Maschine durch eine verschlüsselte Verbindung an den entfernten Server weitergeleitet. Damit steht dem täglichen Umgang mit der PostgreSQL nichts mehr im Wege. Zwar wird MySQL gern wegen seiner Performance gerühmt, wir konnten jedoch keine nennenswerten Unterschiede feststellen. Im Gegenteil: durch den gezielten Einsatz von Stored Procedures konnten wir hier in vielen Bereichen eine Performance Verbesserung erreichen. Andere Dinge die man während der Arbeit mit MySQL lieb gewonnen hat, wie z.B. `auto_increment` Werte, lassen sich spielend durch Sequenzen ersetzen. Hat man also bisher so etwas wie:

```
create table test (id int(11) not null
   auto_increment, ...
```

verwendet, kann man dies jetzt problemlos mit:

```
create table test (id serial not null, ...
```

auf der Postgres erreichen. Durch Verwendung des Typen `serial` wird im Datenbanksystem implizit zur Tabelle noch eine Sequenz erzeugt, die automatisch den nächsthöheren Wert ermittelt.

Weiteres

Das Zend Framework bietet aber auch noch viele andere schöne Dinge, die man bei der Entwicklung großer Projekte schnell mal brauchen kann. So steht mit `Zend_Acl` z.B. eine mächtige, wenn auch nicht triviale, Möglichkeit zur Verfügung Zugriffskontrolllisten zu verwalten. Damit ist es dann möglich die Applikation mit einer umfangreichen Rollenverwaltung auszustatten. Multilingualität der Applikation lässt sich vergleichsweise schnell einbauen. Wir haben hier eine zentrale Klasse definiert, die sich die jeweiligen Übersetzungen aus der Datenbank holt. Diese wird dann wieder per:

```
Zend_Registry::get('lang');
```

im Controller zur Verfügung gestellt. Ein einzelner Eintrag wird dann einfach mit:

```
$this->lang->word('beispiel') ;
```

in die jeweils benötigte Sprache (die wir beim Login ermittelt haben) übersetzt.

Fazit

Rückblickend haben wir durch den konsequenten Einsatz des Zend Frameworks und die damit verbundenen durchgehend objektorientierte Entwicklung, gerade bei der Arbeit im Team eine deutliche Verbesserung in der Geschwindigkeit und der Qualität der Entwicklung zu spüren bekommen. Bereits nach kurzer Eingewöhnungszeit in das Zend Framework kommt man zu sehr ansehnlichen Ergebnissen. Im weiteren Verlauf der Arbeit tritt dann unweigerlich der Punkt ein, an dem man es nicht mehr missen möchte.

Zum Start mit dem Framework und dem Verständnis des Aufbaus der Model-, View-, Controller-Struktur, sei nochmals auf die Mini-Applikation auf der Heft-CD verwiesen. Sie kann aber auch von unserer Website [3] heruntergeladen werden.

Hat man ersten Gehversuche hinter sich, kann man sich getrost auf die eigentliche Arbeit, die Entwicklung der Applikation, konzentrieren und sich ansonsten aus dem reichhaltigen Fundus des Frameworks bedienen. Den Umstieg auf die PostgreSQL können wir im Nachhinein nur empfehlen. Hierin findet man ein ausgezeichnetes, stabiles und überraschend einfach zu bedienendes Datenbanksystem.

Im Internet

- [1] *http://framework.zend.com/home;*
- [2] *http://framework.zend.com/manual/de/;*
- [3] *http://www.e-unique.com/php-solutions.html;*
- [4] *http://framework.zend.com/manual/de/zend.view.html;*
- [5] *http://www.postgresql.org/docs/;*
- [6] *http://www.phpmyadmin.net;*
- [7] *http://phppgadmin.sourceforge.net/;*
- [8] *http://www.pgadmin.org/.*

Listing 7. *Klasse zum Handling der Transaktionen*

```
1   class DbConnector {
2
3   public $dbObj = null;
4   public $selObj = null;
5   public $statements = array();
6   public $error = '';
7
8   public function __construct() {
9      $params = array ('host'      => DB_HOST,
10            'username'   => DB_USER,
11            'password'   => DB_PASS,
12            'dbname'      => DB_NAME,
13            );
14
15     $this->dbObj = Zend_Db::factory(DB_DRIVER, $params);
16  }
17
18  public function sqlAdd($qs) {
19     array_push($this->statements, $qs);
20     return count($this->statements);
21  }
22
23  public function transaction() {
24     $this->dbObj->beginTransaction();
25     try {
26        while ($qs = array_shift($this->statements)) {
27           $this->dbObj->query($qs);
28        }
29        $this->dbObj->commit();
30        $status = 1;
31     } catch (Exception $e) {
32        $this->dbObj->rollBack();
33        $this->error = $e->getMessage();
34        $status = 0;
35     }
36     $this->statements = array();
37     return $status;
38  }
39  }
```

THOMAS PFEIFFER
ANDREAS WENK

Sie arbeiten als Applikations-Entwickler bei der NMMN – New Media Markets & Networks GmbH in Hamburg und sind verantwortlich für die Entwicklung des Raum- und Ressourcenmanagement-Systems eUNIQUE.