PHP-DB2-YUI-ZF Meeting

Selten kommt nur eine Bibliothek alleine zum Einsatz, um ein Projekt zu realisieren. Richtig spannend wird es jedoch, wenn mehrere aktuelle Komponenten wie das Zend Framework, Smarty-Template und die YUI mit der IBM DB2 Datenbank zusammen arbeiten sollen. Wie es trotzdem funktioniert, zeigt folgender Artikel.

von Thomas Wiedmann

Kaum ein Projekt ist klein genug, dass es nicht wenigsten mehrere Bibliothek plus Datenbank benötigt. Getrieben von den Anforderungen des Web 2.0 ist auch mindestens ein JavaScript Framework dabei. Das Ganze dann auch noch unter einen Hut zu bringen, bedarf einiges an Forschungsarbeit, insbesondere wenn nicht die „üblichen Verdächtigen“ aus dem LAMP oder WAMP Umfeld eingesetzt werden sollen, sondern Spezialisten aus dem jeweiligen Fachgebiet. Richtig interessant wird es dann noch, wenn Standards wie XHTML, CSS, XML vorgeschrieben sind. In folgendem Praxisartikel möchte ich Konzepte und Wege aufzeigen, wie

- PHP 5.2.5 [1] plus PECL 5.2.5
- Zend Framework 1.5.0 [2]
- Smarty-Template 2.6.18 [3]
- Yahoo! User Interface Library 2.5.1 [4]
- IBM DB2 Express-C 9.1 [5]
- Apache HTTP Server 2.2.4 [6]

nahezu perfekt zusammen arbeiten. Alle die hier aufgelisteten Komponenten sind entweder Open Source und nach *PHP License, LGPL, BSD* bzw. *Apache License* verfügbar oder kostenfrei nutzbar, wie die IBM DB2 Express-C Datenbank. Die reine Installation der einzelnen Komponenten setze ich voraus, dazu gibt es schon eine Reihe anderer Artikel wie beispielsweise [7][8]. Stürzen wird uns also gleich ins Vergnügen. Auf der Heft-CD finden Sie wie immer den kompletten Source als Zip-Paket inklusive der folgenden Verzeichnisstruktur (Abb. 1).

Diesmal geht es um die Idee zu einem kleinen Testprojekt, wie und ob PHP eine Verbindung zur IBM DB2 aufnehmen kann, von dort native XML-Daten holt und das Ergebnis mit Hilfe von *Smarty* und das YUI *DataTable* möglichst ansprechend anzeigt.

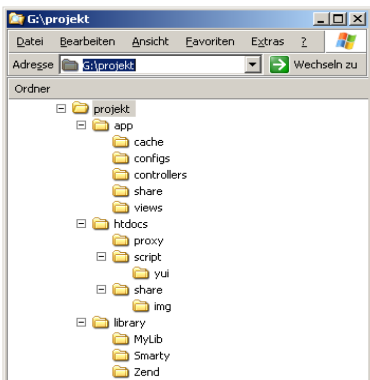


Abb.1: Verzeichnisstruktur für das Zend-Framework MVC-Projekt

Bootstrap – index.php

Das ZF enthält eine Konzeption des Model-View-Controller - kurz MVC-Entwurfsmuster - auf Basis dessen auch unser kleines Projekt konzipiert ist. Das *Model* wird zwar heute etwas zu kurz kommen, aber was nicht ist, kann ja noch werden. Voraussetzung für das MVC-Konzept ist das Rewrite. Damit wird jede Anfrage beim Apache WebSever auf ein bestimmte Startdatei, die sogenannte Bootstrap umgeschrieben also umgeleitet. Für gewöhnlich handelt es sich dabei um die Datei *index.php*, so auch bei uns. Mehr Details dazu erfahren Sie zum Beispiel hier [8]. Unsere *index.php* unterscheidet sich nicht wesentlich von den typischen Konzepten, des Kapitels „*7.1. Zend_Controller Schnellstart*“ im Manual. Das Listing 1 soll deshalb nur kurz die wichtigsten Punkte zum Verständnis aufzeigen:

```
Listing 1:
siehe /htdocs/index.php

[...]
/**
* Konfiguration lesen, Parameter setzen,
* Pfade anpassen
*/
require_once('../app/share/config.php');

/**
* Projektspezifische globale Klasse einbinden
*/
require_once('../app/share/global_class.php');

[...]

/**
* Error-Handler-Plugin abschalten
* (enabled by default)
*/
$frontController->setParam('noErrorHandler',true);

/**
* Action-Helper abschalten (enabled by default)
*/
$frontController->setParam('noViewRenderer',true);

[...]

/**
* Projektspezifische Globale Klasse registrieren
*/
$oGlobal = new global_class();

/**
* Objekt "oGlobal" in die Registry eintragen
*/
Zend_Registry::set('oGlobal', $oGlobal);

[...]
```

Im Listing 1 sind die wesentlichen Teile der *index.php* aufgezeigt. Zuerst werden einige projektspezifischen Parameter und Konstanten geladen und anschließend zwei Standard-Mechanismen des ZF deaktiviert. Besonders der Parameter *noViewRenderer* ist wichtig, damit uns – und damit *Smarty* – das integrierte *Zend_View* nicht frühzeitig in die Quere kommt. Abschließend wird noch eine eigene Klasse *global_class* instanziiert und mit Hilfe *Zend_Registry* den nachfolgenden Controllern zur Verfügung gestellt.

Smarty und Zend View

Das *Smarty* Framework hat schon einige Stürme erlebt, kein Wunder dass es flexibel genug ist, sich in das Zend MVC Konzept einzuklinken und *Zend_View* teilweise zu ersetzen, als auch dessen Features zu nutzen. Beginnen wir also damit, die Klasse *Zend_View* um die *Smarty*-Funktionalität zu erweitern. In unserem eigenen Verzeichnis *MyLibs* stehen alle spezifischen Erweiterungen zu *Zend_View*. Das Listing 2 zeigt den Einstieg.

```
Listing 2:
siehe /library/MyLib/View.php

<?php
...

/**
* Zend_View Klasse laden
*/
require_once ('Zend/View.php');

/**
* Smarty Library einbinden
*/
require_once(PROJEKT_SMARTY.'libs/Smarty.class.php');

/**
* Zend - Registry einbinden
*/
require_once ('Zend/Registry.php');


/**
* Erweiterung der Basisklasse: Zend_View_Abstract
* um die Smarty-Funktionalität
*/
class MyLib_View extends Zend_View
[...]
```

Dabei entsteht die neue Klasse *MyLib_View*. Wichtig ist dabei, dass das Skript *Smarty.class.php* – also die eigentliche *Smarty-Engine* - eingebunden wird. Hierzu kommt eine globale Konstante ins Spiel, die einen absoluten Pfad zum *Smarty-Framework* enthält. Definiert wird diese und noch ein paar weitere Konstanten in /app/share/config.php. Diese Konfigurationen sind bereits im Bootstrap *index.php* geladen worden und stehen somit hier zur Verfügung. Damit kann es mit Listing 3 weiter gehen.

```
Listing 3:
siehe /library/MyLib/View.php

[...]
public function __construct($data = array()) {

 parent::__construct($data);
 $this->_smarty = new smarty();

 /**
 * Smarty eigene Verzeichnisse/Einstellungen
 * View-Path wird später zur Laufzeit vorgegeben,
 * siehe Funktion: render()
 */
 $this->_smarty->template_dir = '';
 $this->_smarty->compile_dir = PROJEKT_SMARTY.'templates_c';
```

```
 $this->_smarty->config_dir = PROJEKT_SMARTY.'configs';
 $this->_smarty->cache_dir = PROJEKT_SMARTY.'cache';

 /**
 * Muster für Smarty Template Variablen
 * und Funktionen
 */
 $this->_smarty->left_delimiter = '{{';
 $this->_smarty->right_delimiter = '}}';
[...]
```

Im Konstruktor von unserem *View.php* wird eine *Smarty*-Instanz erzeugt und die Smarty-spezifischen Einstellungen festgelegt. So oder ähnlich werden es die meisten *Smarty*-Nutzer wieder erkennen. Wichtig ist wiederum die Konstante PROJEKT_SMART, die den absoluten Pfad zu den *Smarty*-eigenen Verzeichnissen definiert. Beachten Sie bitte noch die beiden Delimiter (*left, right*) für die *Smarty*-Template-Variablen. Für die doppelte geschweifte Klammer habe ich mich deshalb entschieden, damit wir später nicht mit eventuellen CSS Definitionen in Konflikt kommen, die eine einfache geschweifte Klammer verwenden. Natürlich können Sie auch andere Zeichen zur Mustererkennung hier festlegen, doch manchmal wird es richtig schwierig, etwas eindeutiges zu finden. Weiterhin werden in unserer *View.php* noch folgende *Zend_View* Methoden ergänzt oder überschrieben:

- _run()
- assign()
- escape()
- isCached()
- setCaching()
- render()

In die letztere Methode - nämlich *render()* - lohnt es sich einen kurzen Blick zu werfen (Listing 4).

```
Listing 4:
siehe /library/MyLib/View.php

[...]
public function render($name)  {

 /**
 * Smary-Variable "template_dir" mit dem Path zu
 * den Template füllen,
 * dies ist notwendig, wenn innerhalb der Smarty
 * Templates
 * mit {{include file="footer.tpl"}} Befehlen
 * gearbeitet wird
 */
 $this->_smarty->template_dir = $this->getScriptPath(null);

 /**
 * jetzt folgt der Aufruf von
 * Zend_View_Abstract->render() und von dort
 * aus $this->_run().
 */
 print parent::render($name);

}
```

In den jeweiligen Controllern wird später der Name des Templates an die Methode *render()* übergeben. Der Pfad in welchem Verzeichnis dieses Template letztlich liegt, wird

erst zu diesem Zeitpunkt *Smarty* mitgeteilt, damit kann die volle Flexibilität der Skriptpfade von *Zend_View* genutzt werden. Nun geben wir die Steuerung wieder ab an das *Zend_View_Abstract*, indem das Programm dessen *render()* Methode aufruft. Der Rückgabewert von *parent::render()* ist dann auch schon das ausgewertete und fertig ausgefüllte *Smarty-Template*. Bis es allerdings soweit ist, sind noch ein paar Hürden zu nehmen, die im Folgenden aufgezeigt werden. Werfen wir nun ein Blick auf das „C“ von MVC, spricht den Controller.

IndexController

Ist alles korrekt installiert und läuft der lokale Apache WebServer, können wir in unserem Browser die URL http://localhost eingeben. Diese Anfrage landet aufgrund des aktivierten Rewrite unweigerlich bei *index.php* und wird mit Hilfe des ZF-Front-Controllers, an den ZF-Dispatcher übergeben. Dieser analysiert den Request und leitet schließlich an den voreingestellten Default Controller *indexController.php* und Default Action i*ndexAction* weiter. Mit Listing 5 werfen wir einen ersten Blick darauf:

```
Listing 5:
siehe /app/controllers/IndexController.php

[...]
Zend_Loader::loadClass('Zend_Controller_Action');
Zend_Loader::loadClass('MyLib_View');

class IndexController extends Zend_Controller_Action
{
 /**
 * View/Smarty Template Objekt innerhalb diesen
 * Controller.
 * @var object
 */
 protected $oView = null;

 /**
 * Initialize View/Smarty object
 * Der Aufruf von init() erfolgt aus
 * Zend/Controller/Action.php heraus.
 * Diese Methode muss "public" sein.
 */
 public function init() {
  $this->oView = new MyLib_view(array('scriptPath' => BASEPATH.'app/views'));
 }
[...]
```

Gleich zu Beginn werden unsere Smarty-Anpassungen *MyLib_View* geladen, die wir bereits weiter oben (Listing 2+3) besprochen haben. Mit dem automatischen Aufruf der Initialisierungs-Methode *Init()* wird das neue *Smarty*-View-Objekt instanziiert und gleichzeitig der Pfad zu den Templates eingetragen. Mit Hilfe des Objektes *$this->oView* haben wir anschließend Zugriff auf die *Smarty* und die *ZF-View* Methoden. In Listing 6 fügen sich alle bisherigen Vorbereitung zusammen für GUI-Ergebnis – das „V“, also die View aus dem MVC-Konzept.

```
Listing 6:
siehe /app/controllers/IndexController.php
[...]

public function indexAction() {

 /**
 * Globale Klasse aus der Registry holen
 */
 $oGlobal = Zend_Registry::get('oGlobal');

 /**
 * HTML/HTTP/CSS Grundeinstellungen- und
 * Parameter in das Smarty-Template
 * eintragen
 */
 $this->oView->assign('Doctype', $oGlobal->getDoctype());
 $this->oView->assign('Titeltext', 'PHP-DB2-YUI Meeting');
 $this->oView->assign('Metadaten', $oGlobal->getMetadaten());

 /**
 * Smarty Template ausfüllen und das Ergebnis
 * an den Browser schicken
 */
 echo $this->oView->render('index.tpl');

}
```

Im ersten Schritt holen wir aus *Zend_Registry* das globale Objekt *$oGlobal.* Im Bootstrap (Listing 1) wird es in die *Zend_Registry* geschrieben und ZF hat es für uns frisch gehalten. Mit Hilfe von *$oGlobal* haben wir Zugriff auf unsere projektspezifischen Methoden, die nun beim Ausfüllen (*assign*) des Smarty-Templates zum Einsatz kommen. Hinter der Methode *render* verbergen sich die *Smarty-Engine* und Teile von *Zend_View,* mit Hilfe derer das Template (i*ndex.tpl*) so bereitet wird, dass es schlussendlich an den Browser ausgegeben werden kann. Ein kurzer Blick in den Aufbau des Templates (Listing 7), zeigt wie die Smarty-Templates-Tags – mit den doppelten „{{„ und „}}“ - in den HTML-Code integriert sind.

```
Listing 7:
siehe /app/views/index.tpl

{{$Doctype}}
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de" lang="de">
 <head>
   <title>{{$Titeltext}}</title>
   {{$Metadaten}}
 </head>
 <body>
  <div id="doc3" >
   <div id="bd">
   <h1>Hello PHP-World!</h1>
   <h3>mit Zend-Framework, Smarty-Templates, YUI-DataTable und IBM DB2</h3>
   <fieldset><legend>Funktionsauswahl</legend>
    <ul>
      <li><a href="/db2/login">DB2 Login für den Verbindungstest</a></li>
      <li><a href="/xml/login">DB2 Login für den XML/JSON-Test</a></li>
    </ul>
   </fieldset>
   </div>
  </div>
 </body>
</html>
```

Am Browser sieht das aufbereitete Ergebnis dann – hoffentlich auch bei Ihnen – in etwa so aus (Abb. 2).

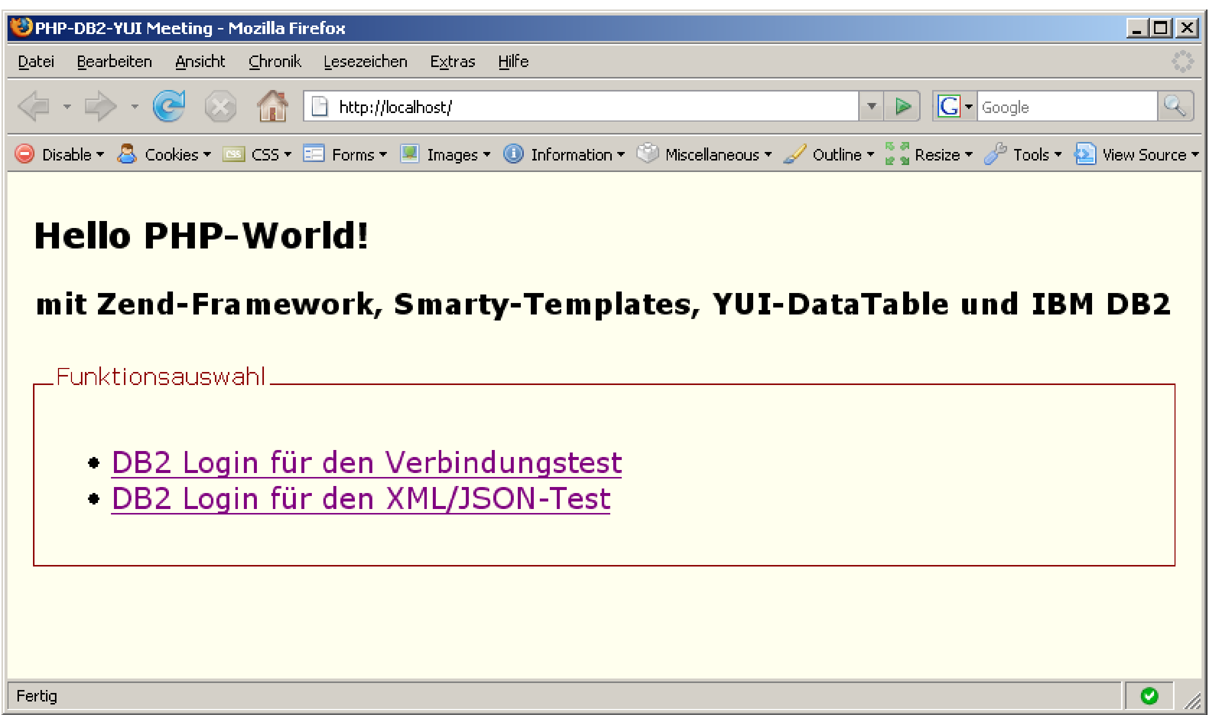


Abb.2: Hello „PHP-World“ mit Hilfe von ZF und Smarty.
Standards, CSS und YUI–Grid

Das hinter dem recht schlichten Smarty-Template (Listing 7) doch mehr versteckt ist, zeigt ein Blick hinter die Kulissen. Es stellt sich beispielsweise die Frage, woher die Cascading-Styles-Sheets (CSS) zur Gestaltung der Seite eigentlich kommen? Verdächtig macht sich das Smarty-Tag {{$Metadaten}}. Hier können notwendige Styles zur Laufzeit eingefügt werden. Mit Hilfe des kostenlosen Firefox-Add-On „Firebug“ [9] – ein absolut empfehlenswertes Entwicklerwerkzeug - kommen Sie mir schnell auf die Schliche (Abb. 3).

Inspect Clear All HTML CSS JS XHR Images Flash
Console HTML CSS Script DOM Net Options

localhost	localhost	2 KB	0ms
reset-min.css	localhost	849 b	0ms
base-min.css	localhost	750 b	16ms
grids-min.css	localhost	5 KB	0ms
style.css	localhost	850 b	15ms
5 requests		9 KB	31ms

Abb.3: Firebug zeigt welches CSS-Files geladen wird.

Insgesamt werden fünf Anfragen abgearbeitet. Neben *localhost* – dem eigentlichen HTML-Gerüst der Seite – folgen insgesamt vier CSS-Dateien für die optischen Einstellungen. Wobei *reset-min.css* und *base-min.css* versuchen alle browserspezifischen CSS-Einstellungen auf YUI-Definitionen zu setzen. Grundlage hierfür ist die *Yahoo!'s philosophy*

of Graded Browser Support für die unterstützen Browsertypen [10]. Das Seitenlayout wird von *grids-min.css* definiert. Mit Hilfe dieser gerade mal 5kB großen Datei lassen sich eine Menge flexibler Grid-Layouts festlegen. Diese drei Dateien sind ein Teil der *Yahoo! User Interface Library* kurz YUI, dazu später noch mehr. Mit *styles.css* kommen dann noch die projektspezifischen Einstellungen ins Spiel.

DB2 Login für den Verbindungstest

IBM stellt mit der DB2 Express C eine kostenlose und nahezu uneingeschränkte Version ihrer Datenbank zur Verfügung. Die Community hat bisher kaum wahrgenommen, welche RDBMS-Power und nun auch *pureXML* Leistungsqualität hier kostenlos zur Verfügung steht. Auch mit ihren Limitierungen, kann die DB2 Express-C problemlos Software für ganze Abteilungen bedienen. Mit dem Zend Core for DB2 [11] ist zudem ein vollständiges Paket verfügbar, welches den Einstieg erleichtern soll. Aber vielleicht liegt es auch an der verwirrenden Vielfalt von PHP Treibern, die einen *Connect* zur DB2 Express-C Datenbank ermöglichen. Insgesamt fünf Varianten

- IBM DB2 (siehe PECL)
- ODBC
- PDO-ODBC
- PDO-ODBC (UC)
- PDO-IBM (siehe PECL)

stehen zur Verfügung und werden im Folgenden besprochen. Die erste und die letzte Verbindungstechnik - in der Auflistung - bieten einen nativen Zugriff auf die DB2-Datenbank und was für einen Vorteil dies hat, darauf kommen wir noch etwas später zu sprechen.

Mit einem Klick auf den Link *DB2 Login für den Verbindungstest* (Abb. 2) starten wir den Anmeldedialog im *Db2Controller.php* und dort die *loginAction.* Hierzu gleich noch den Hinweis*:*Ab der Version ZF 1.5.0 müssen die Action-Namen klein geschrieben sein, sonst kann der Dispatcher diese nicht mehr finden. Wer dies deaktivieren möchte oder muss, kann im Bootstrap (index.php) folgenden Parameter setzen (Listing 8):

```
Listing 8:
siehe /htdocs/index.php

[...]
$frontController->setParam('useCaseSensitiveActions', true);
[...]
```

Der Login-Dialog für die DB2-Anmeldung (Abb. 4) enthält eine Reihe von Eingaben, die notwendig sind, um die verschiedenen Möglichkeiten des Verbindungsaufbau testen zu können.

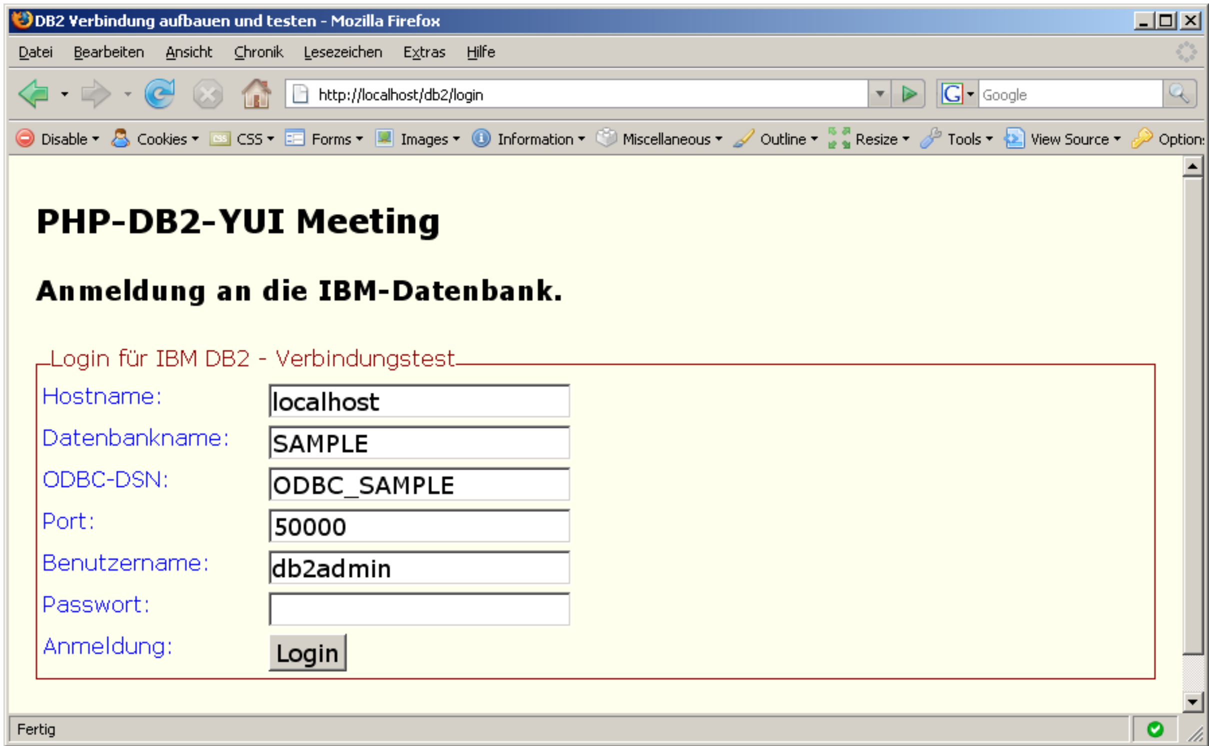
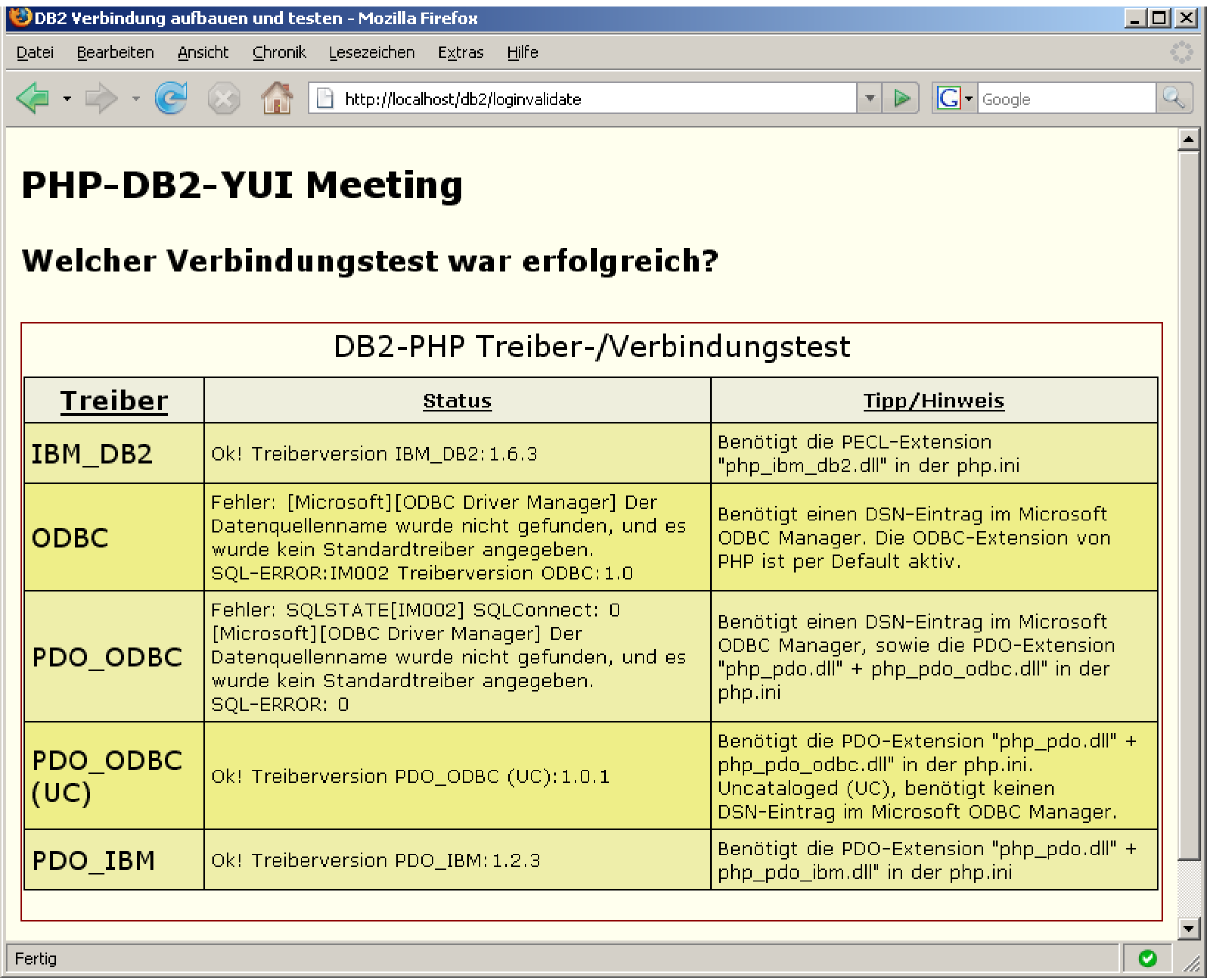


Abb.4: Anmeldedialog zum DB2-Verbindungstest.

Wir gehen davon aus, dass der DB2-Server identisch zu *localhost* ist. Bei einer typischen DB2-Installation kann eine Testdatenbank *SAMPLE* angelegt werden. Damit die DB2 *pureXML* Technologie [12] richtig genutzt werden kann, muss *SAMPLE* mit dem Zeichencode UTF-8 erzeugt werden.

```
CREATE DATABASE sample USING CODESET utf-8 TERRITORY DE
```

Dies soll bei unserem Test vorerst genügen. Per Default lauscht die DB2-Instanz auf dem Port *50000*. Als Datenbankadministrator wird zudem ein Benutzerkonto *db2admin* angelegt. Bleibt noch der ODBC-DSN. Unter Windows kann auch der MS-ODBC Manager genutzt werden. Wird die SAMPLE Datenbank katalogisiert, erfolgt unter anderem auch ein Eintrag im MS-ODBC Manager (odbc.ini und db2cli.ini). Diesen ODBC Data-Source-Name (DSN) nenne ich *ODBC_SAMPLE*, um darzustellen, dass die Verbindung erst über *ODBC_SAMPLE* zur eigentlichen Datenbank *SAMPLE* findet. Auch ein *uncataloged* ODBC-Zugriff ist möglich, wie Sie gleich sehen werden. Aber geben wir das gültige Passwort ein und führen Verbindungstest aus. Das Ergebnis kann dann so aussehen (Abb. 5).


PHP-DB2-YUI Meeting

Welcher Verbindungstest war erfolgreich?

DB2-PHP Treiber-/Verbindungstest

Treiber	Status	Tipp/Hinweis
IBM_DB2	Ok! Treiberversion IBM_DB2:1.6.3	Benötigt die PECL-Extension "php_ibm_db2.dll" in der php.ini
ODBC	Fehler: [Microsoft][ODBC Driver Manager] Der Datenquellenname wurde nicht gefunden, und es wurde kein Standardtreiber angegeben. SQL-ERROR:IM002 Treiberversion ODBC:1.0	Benötigt einen DSN-Eintrag im Microsoft ODBC Manager. Die ODBC-Extension von PHP ist per Default aktiv.
PDO_ODBC	Fehler: SQLSTATE[IM002] SQLConnect: 0 [Microsoft][ODBC Driver Manager] Der Datenquellenname wurde nicht gefunden, und es wurde kein Standardtreiber angegeben. SQL-ERROR: 0	Benötigt einen DSN-Eintrag im Microsoft ODBC Manager, sowie die PDO-Extension "php_pdo.dll" + php_pdo_odbc.dll" in der php.ini
PDO_ODBC (UC)	Ok! Treiberversion PDO_ODBC (UC):1.0.1	Benötigt die PDO-Extension "php_pdo.dll" + php_pdo_odbc.dll" in der php.ini. Uncataloged (UC), benötigt keinen DSN-Eintrag im Microsoft ODBC Manager.
PDO_IBM	Ok! Treiberversion PDO_IBM:1.2.3	Benötigt die PDO-Extension "php_pdo.dll" + php_pdo_ibm.dll" in der php.ini

Abb.5: Ergebnis des DB2-Verbindungstest als YUI *DataTable*.

Insgesamt werden fünf verschiedene Techniken eingesetzt, um eine Verbindung zur DB2 herzustellen. Funktioniert ein Test, wird in der Spalte „Status“ die Treiberversion der jeweilig eingesetzten Technik ermittelt. Kommt keine Verbindung zu Stande, wie beispielsweise bei den beiden ODBC-Varianten, dann wird Fehlermeldung ausgegeben. Grund für den Fehler ist der nicht vorhandene Eintrag im ODBC-Manager. Das es auch ohne diesen Eintrag funktionieren kann, zeigt die Variante *PDO_ODBC (UC)*, die im Prinzip identisch zu *PDO_ODBC* ist, bis auf den Unterschied, dass beim Connect ein kompletter DSN-String übergeben wird (Listing 9). Dies ist dann notwendig, wenn bei der Installation des DB2 Client kein Eintrag in den MS-ODBC Manager erfolgt ist. Jede Spaltenüberschrift der Ergebnistabelle (Abb. 5) ist unterstrichen, dies bedeutet, der Tabelleninhalt wird bei einem Klick auf eine dieser Spaltenüberschriften automatisch sortiert und zwar *client*-seitig, ohne eine erneute Anfrage beim Web-Server. Damit so etwas funktionieren kann, bedarf es JavaScript und in unserem Beispiel das *DataTable* von YUI. Wie das Ganze funktioniert lesen Sie etwas später, vorerst schauen wir uns die Techniken für den Verbindungsaufbau zur DB2 Datenbank beispielhaft an *PDO-ODBC (UC)* und *PDO-IBM* an.

```
Listing 9:
siehe /app/controllers/Db2Controller.php

[...]
private function testpdoodbc2() {

 /**
 * Globale Klasse aus der Registry holen
 */
 $aFormValues = Zend_Registry::get('aFormValues');

 try {
  /**
  * PDO_ODBC (UC=Uncataloged)  Datenbankverbindung
  *
  */
  $dbh = new PDO(
    'odbc:DRIVER={IBM DB2 ODBC DRIVER};'.
    'HOSTNAME='.$aFormValues['df_sHostname'].';'.
    'PORT='.$aFormValues['df_nPort'].';'.
    'DATABASE='.$aFormValues['df_sDatabase'].';'.
    'PROTOCOL=TCPIP;'.
    'UID='.$aFormValues['df_sUser'].';'.
    'PWD='.$aFormValues['df_sPassword']
  );

  if ($dbh) {
   $dbh = null;
   return 'Ok! Treiberversion PDO_ODBC (UC):'.phpversion('pdo_odbc');
  } else {
   return 'Fehler: $dbh ist null. Treiberversion
             PDO_ODBC (UC):'.phpversion('pdo_odbc');
  }
 } catch (PDOException $e) {
  return 'Fehler: '. $e->getMessage().'. SQL-ERROR: '.$e->getCode();
 }
}
[...]
```

In Listing 9 wird das PHP Data Object (PDO) für den Connect verwendet. Als Parameter wird das ausführliche Format des Data-Source-Name (DSN) genutzt. Wie in Abb. 5 – *PDO_ODBC (UC)* zu sehen ist, kann damit auch eine nicht im MS-ODBC Manager eingetragene Datenquelle genutzt werden. Kann PDO die Verbindung aufbauen, erhalten wir dafür als Rückgabewert einen Datenbank-Handle in Form der Variablen *$dbh.* Klappt der Verbindungsaufbau nicht, wirft PDO einen Fehler und dieser wird mit Hilfe des *catch* –Blockes gefangen und ausgewertet. Ohne den MS ODBC-Manager kommen die Varianten *IBM_DB2* und *PDO_IBM* aus (Listing 10). Beide Varianten können über die PECL Distribution bezogen werden [1], die zweite sollte dann berücksichtigt werden, wenn ein eher Datenbankhersteller unabhängiges Konzept wichtig ist.

```
Listing 10:
siehe /app/controller/Db2Controller.php

[...]
try {

 /**
 * PDO_IBM Datenbankverbindung
 */
 $dbh = new PDO('ibm:'.
                $aFormValues['df_sDatabase'],
```

```
                  $aFormValues['df_sUser'],
                  $aFormValues['df_sPassword'],
                  array());

 if ($dbh) {
  $dbh = null;
  return 'Ok! Treiberversion PDO_IBM:'.phpversion('pdo_ibm');
[...]
```

In Listing 10 ist ein Beispiel für den Verbindungsaufbau zu sehen. *PDO-IBM* benötigt lediglich vier Parameter und zwar den PDO Datenbanktreiber „ibm“, Datenbanknamen „sample“, sowie Benutzer und Passwort. Alternativ können als Array noch spezifische Parameter übergeben werden.

YUI DataTable und Smarty

Im vorherigen Abschnitt (Abb. 5) bin ich bereits auf das YUI *DataTable* zu sprechen gekommen, mit Hilfe dessen die Tabelle und das Sortieren der Spalten realisiert ist. Das YUI *DataTable* ist eines der wichtigsten und auch mächtigsten Objekte des YUI-Framework. Leider hat es immer noch „Beta“ Status, was sich in regelmäßigen in kleineren API-Änderungen äußert und deshalb häufig eigene Anpassungen notwendig macht. Inwieweit die aktuelle Version 2.5.1 stabil genug ist, wird sich zeigen. Trotz allem ist die verfügbare Funktionalität sehr umfangreich. Hinter YUI stehen unter anderem zehn YAHOO-Entwickler und eine aktive Community. Vor kurzem wurde in der Newsgroup die YUI-Roadmap für die nächsten Versionen vorgestellt, deren Planung reicht bis weit über die 3.0 Version hinaus. Diese Power garantiert stete Weiterentwicklungen und Verbesserungen. Werfen wir nun einen Blick auf das Smarty-Template mit integriertem YUI-*DataTable* (Listing 11).

```
Listing 11:
siehe /app/views/db2phpresult.tpl

[...]
<script type="text/javascript"
  src="/script/yui/build/datasource/datasource-beta-min.js"></script>
<script type="text/javascript"
  src="/script/yui/build/datatable/datatable-beta-min.js"></script>

<link rel="stylesheet" type="text/css"
  href="/script/yui/build/datatable/assets/skins/sam/datatable.css" />
[...]
```

Typischerweise benötigt man bei einem Einsatz von *Rich Internet Applications* (RIA) eine Reihe von JavaScript-Dateien, die der Browser erst einmal „mit einem tiefen Schluck“ laden muss. Sind die notwendigen Skripte aber einmal lokal vorhanden, können alle weiteren Aufrufe mit Hilfe des browsereigenen Cache zufrieden gestellt werden und die eventuelle Geschwindigkeitseinbuße gering halten. In Listing 11 werden eine Reihe von YUI-Libraries geladen, einige sind - je nach Anwendung und gewünschter Funktionalität – optional. Die Abhängigkeiten zwischen den einzelnen Libraries sind sowohl in der Dokumentation, als auch direkt im Source beschrieben. Beim *DataTable* beispielsweise *@requires yahoo, dom, event, element, datasource @optional connection, dragdrop.* Mit dem Erscheinen der YUI 2.5.1 hat Yahoo der Entwicklergemeinde den *YUI Dependency Configurator* spendiert, mit Hilfe dessen die Abhängigkeiten der einzelnen Module zu den JavaScript-Libraries ermittelt

werden können. Um die Netzbelastung möglichst gering zu halten, gibt es die Library *utilities.js.* Dieses beinhaltet und kombiniert die häufig verwendeten Komponenten

- Yahoo Global Object
- Event
- Dom
- Connection Manager
- Animation
- Drag & Drop
- Element
- Get
- YUI Loader

Das Listing 12 zeigt, wie Sie die Darstellung des YUI-*DataTable* mit Hilfe CSS individuell gestalten können.

```
Listing 12:
siehe /app/views/db2phpresult.tpl

[...]
<style type="text/css">
/* Eigene CSS für YUI DataTable */
#json {margin:0.1em; }
#json table {border-collapse:collapse; font-size:0.9em; color:#000000;
             background-color:#EEEEAA;}
#json th, #json td {padding:.2em; border:1px solid #000;}
#json th {background-color:#EEEEDD; }
#json th a {color:#000000;}
#json th a:hover {color:#000000; }
#json th a:visited {color:#000000; }
#json th a:focus {color:#000000; }
#json .yui-dt-odd {background-color:#EEEE88;}
#json .yui-dt-col-Hint {font-size:0.7em;}
#json .yui-dt-col-Status {font-size:0.7em;}
[...]
```

Das YUI *DataTable* Objekt wird in einem DIV-Block *<div id="json"></div>* zur Laufzeit erzeugt. Auf diese *id =“json“* beziehen sich die individuellen CSS Anpassungen - also *#json* - plus dem jeweiligen HTML Element. Die individuellen CSS Anpassungen überschreiben die Standarddefinitionen des YUI-eigenen CSS. Hierbei ist das eigentliche Problem herauszufinden, welche CSS-Definition denn nun überschrieben werden muss, um das gewünschte zu erreichen. Das Firebug Add-On [9] ist dabei eine nahezu unverzichtbare Hilfe. Damit kommen wir zum JavaScript-API-Teil, dem Layout und Aufruf von YUI *DataTable* (Listing 13).

```
Listing 13:
siehe /app/views/db2phpresult.tpl

[...]
/**
* YUI DataTable
*/
YAHOO.util.Event.onDOMReady(function() {
  YAHOO.example.json = new function() {
    var myColumnDefs = [
        {key:"Driver", label:"Treiber", sortable:true, minWidth:"180" },
        {key:"Status", label:"Status", sortable:true },
```

```
        {key:"Hint", label:"Tipp/Hinweis", sortable:true }
    ];

    /**
    * JSON Daten füllen
    */
    var sDataJSON = {{$sDataJSON}};

    this.myDataSource = new YAHOO.util.DataSource(sDataJSON);
    this.myDataSource.responseType = YAHOO.util.DataSource.TYPE_JSON;
    this.myDataSource.responseSchema = {
        resultsList: "ResultSet.Result",
        fields: ["Driver","Status","Hint"]
    };
[...]
```

Mit *YAHOO.util.Event.onDOMReady()* startet der Aufbau des YUI *DataTable* (Listing 13). Die Variable *myColumnsDefs* definiert alle Tabellenspalten und deren Eigenschaften wie beispielsweise *label*, *sortable* und *minWidth*. Dies sind nur drei von einer Reihe weitere optionaler Eigenschaften, die für die Spalten festgelegt werden können. Besondere Bedeutung kommt allerdings der Eigenschaft *key* zu. *key* beschreibt den eindeutigen, case-sensitiven Spaltennamen, mit Hilfe dessen die übergebenen Daten der Tabellenspalte zugeordnet werden. Mit dem *Template-Tag {{$sDataJSON}}* kommt wieder *Smarty* ins Spiel. Hier wird zur Laufzeit von *Smarty* ein vorbereiteter JSON Datenblock eingesetzt, den YUI dann JSON-Konform auflöst *YAHOO.util.DataSource(sDataJSON).* Sie sehen, es muss nicht immer AJAX sein, das YUI *DataTable* kommt auch mit einem simplen Datenblock zurecht. Ganz wichtig dabei ist der korrekte Aufbau des *responseSchema.* Hiermit wird festgelegt, mit welchen *Tag* die JSON Datenliste beginnt und aus welchen Spalten und exakt welcher Struktur die Daten aufgebaut sind. Die exakte case-sensitive Schreibweise ist Pflicht und muss mit der oben besprochenen Spalte *key* übereinstimmen, wobei nicht unbedingt alle Spalten des *responseSchema* auch als *key* in der Spaltendefinition erscheinen müssen. Beispielweise kann so sehr einfach Informationstext in Form eine Tooltip-Spalte übergeben werden, ohne dass diese sofort als eigene Spalte angezeigt werden muss. Ein Fehler im Aufbau des *responseSchema* ist der häufigste Grund für die unbeliebte YUI Fehlermeldung: *Data error.*

```
Listing 14:
siehe /app/views/db2phpresult.tpl
[...]

var oConfigs = {
    caption:"DB2-PHP Treiber-/Verbindungstest",
        sortedBy:{key:"Driver",dir:"asc"}
};

YAHOO.widget.DataTable.MSG_EMPTY ={{$sMSG_EMPTY}};
this.myDataTable = new YAHOO.widget.DataTable("json",
     myColumnDefs, this.myDataSource, oConfigs);
[...]
```

Weiterhin kann *DataTable* mit Hilfe von *oConfigs* mitgeteilt werden, nach welcher Spalte die Daten bereits vorsortiert (*sortedBy*) sind. Die englischsprachige Meldung „empty table“ kann mittels einem *Smarty*-Tag *{{$sMSG_EMPTY}}* zur Laufzeit an die eigene Sprache angepasst werden. Als letzten Schritt werden alle Daten, Objekte und Einstellungen an das *DataTable Widget* übergeben und damit die Tabelle im DIV-Block *<div id="json"></div>* erzeugt.

DB2 und pureXML Support

Wie bereits eingangs erwähnt, enthält DB2 ab der Version 9.1 eine native XML Speicherung. Die XML-Daten werden also beim Speichern weder in einem BLOB-Feld gespeichert, noch in ein relationales Model aufgetrennt (*Shredding).* Dafür hat IBM ihrem Flagschiff einen neuen Datentyp *XML* spendiert. Eine Tabelle für herkömmliche und XML-Daten sieht beispielhaft so aus (Listing 15):

```
Listing 15:

CREATE TABLE person (
 person_id INT NOT NULL,
 person_info XML NOT NULL,

 PRIMARY KEY (person_id)
);


INSERT INTO person
(person_id , person_info ) VALUES
(1,'<person>
     <titel></titel>
     <anrede>Herr</anrede>
     <vorname>Fritz</vorname>
     <nachname>Muster</nachname>
     <geburtstag>2006-10-21</geburtstag>
     <telefon></telefon>
     <mobiltelefon>017200000000</mobiltelefon>
     <email>fritz@muster.net</email>
    </person>');
```

Die Tabelle *person* hat eine Spalte *person_info* vom Datentyp XML. Diese Spalte kann eine komplette XML-Struktur aufnehmen, die später direkt mittels *XPath* oder auch *XQuery* ausgewertet werden kann. Mit folgendem SQL werden alle Personen heraus gesucht, die nach dem 01.01.1960 geboren sind.

```
Listing 16:

$sql = null;
$sql .= 'SELECT person_id, person_info ';
$sql .= '  FROM person ';
$sql .= ' WHERE XMLEXISTS( ';
$sql .= ' \'$person_info[person/geburtstag>= ';
$sql .= '                    "1960-01-01"]\' ';
$sql .= ' PASSING person_info AS "person_info")';
```

Bitte beachten Sie den Teil *$person_info*. Hierbei handelt es sich nicht – wie vielleicht vermutet – um eine PHP-Variable, sondern um einen Befehlsteil von XMLEXISTS. Um hier PHP vor eventuellen Fehlinterpretationen zu schützen, müssen die String-Inhalte in einfache Hochkommas gestellt werden und nicht etwa in doppelte. PHP analysiert ansonsten solche Inhalte und versucht *$person_info* durch einen Variablenwert zu ersetzen, den es nicht gibt. Diese gemischte Art von SQL und XML ist definiert durch den *SQL:2003 Standard Part 14 SQL/XML*.

Wie pureXML und JSON nutzen?

Nach diesen Vorbereitungen können wir mit unseren PHP-Controllern weiter forschen. Wie in Abb. 2 zu sehen ist, gibt es da einen zweiten Link namens *DB2 Login für den XML/JSON-Test.* Der dahinter liegende Link verweist auf *http://localhost/xml/login* und damit auf Controller *Xml* mit der Action *login*. Lösen Sie den Link aus, sollte folgendes im Browser erscheinen (Abb. 7).

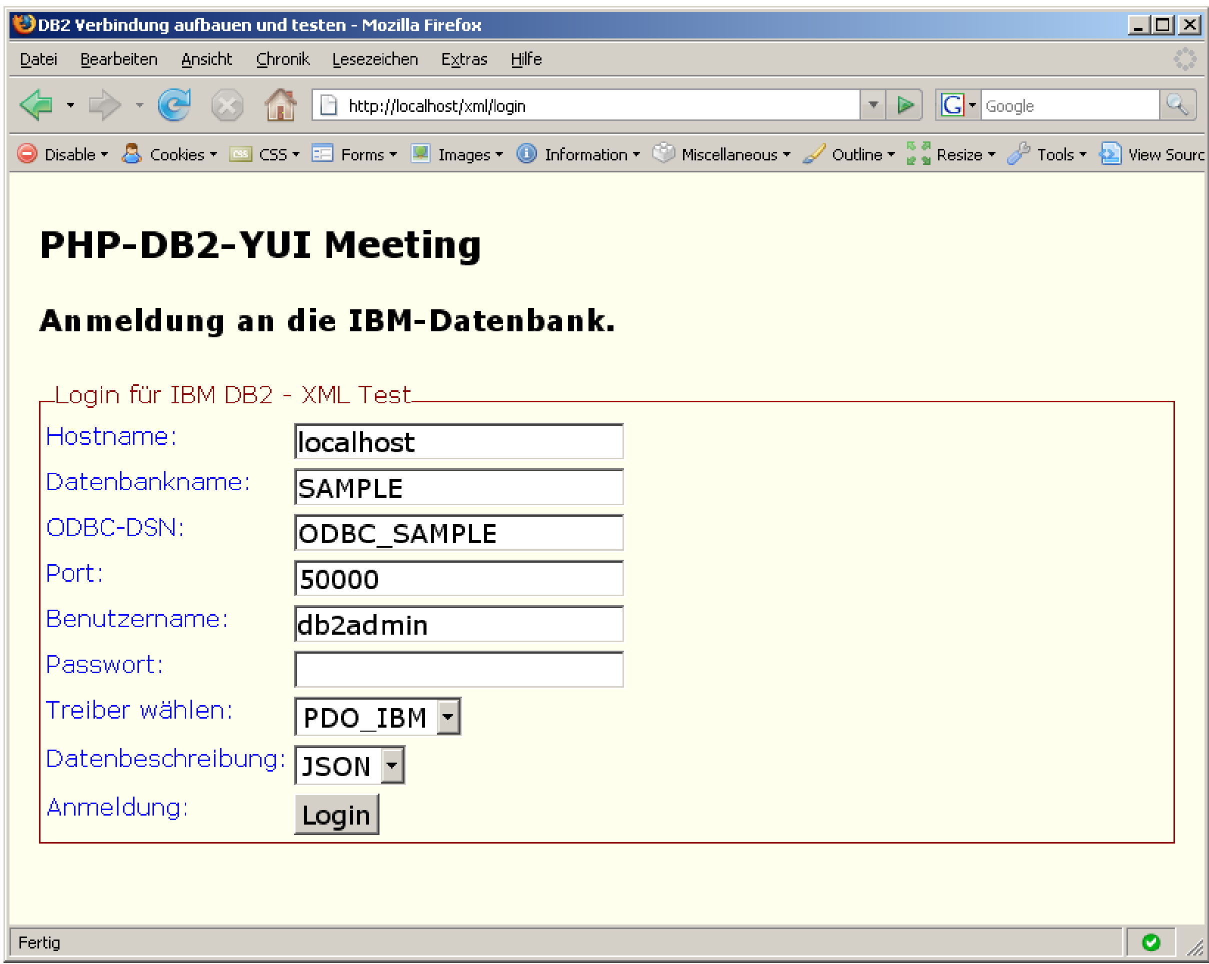


Abb.7: Anmeldedialog zum DB2-XML/JSON-Test.

Beginnen wir mit der Autorisierung an der Datenbank (Abb. 7), sowie mit der Auswahl des PHP-Treibers, über den der Zugriff auf die XML-Tabelle *person* erfolgen soll. Als Treiber stehen hier nur noch IBM_DB2 und PDO_IBM zur Auswahl. Die auf ODBC basierenden Varianten kommen nicht (oder noch nicht) mit dem neuen DB2 Datentyp XML zurecht. Eventuell gibt es für die ODBC Varianten noch spezifische Parameter, um dieses Problem zu beseitigen, aber bei meinen verschiedenen Tests habe ich keine Lösung gefunden. Bleiben also die beiden nativen Treiber zur Auswahl. Prinzipiell ist dies kein Problem, sondern eher ein Vorteil nicht erst den Umweg über die ODBC-Schicht zu nehmen. Unter der Auswahl *Datenbeschreibung* kann JSON oder XML ausgewählt werden. Beide Varianten werden wir im Folgenden durchspielen. Bei der JSON Variante sendet PHP/Smarty die JSON-Daten direkt in den Browser, bei der XML Variante zeige ich ein Beispiel, wie YUI mit Hilfe eines

PHP Proxy auf die Datenbank-Tabelle zugreift und YUI dann die XML-Daten direkt verarbeitet. Beginnen wir mit der JSON Variante. Das Ergebnis der SQL-Abfrage sollte ähnlich der Abb. 8 sein.

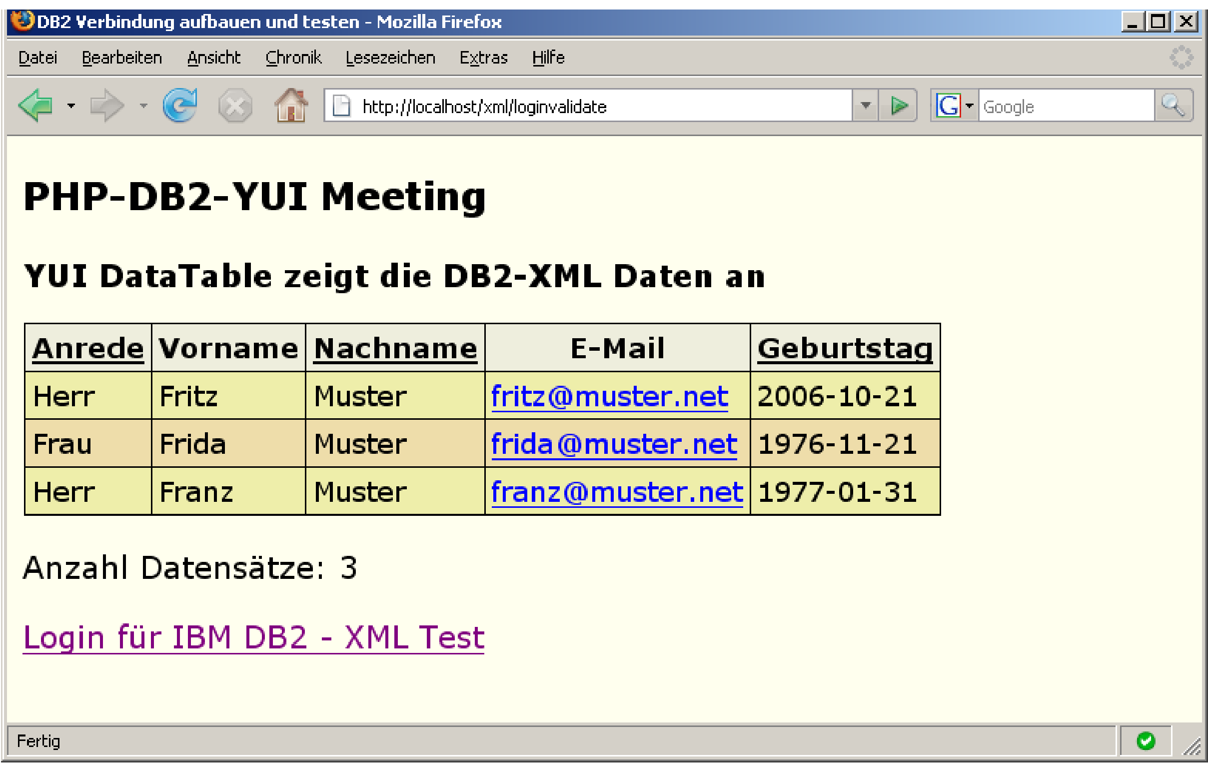


Abb.8: Ergebnistabelle für die JSON Abfrage

Mit Hilfe von YUI *DataTable* wird die Abfrage aus Listing 16 optisch aufbereitet. Zuvor wurde im *XmlController* die Ergebnismenge der Abfrage mit Hilfe der statischen Methode *Zend_Json::fromXml* umgewandelt. Diese praktische *fromXml* Methode, die einen kompletten XML-String in JSON umwandeln kann, steht erst seit der ZF-Version 1.5 zur Verfügung. Der zentrale YUI-Teil zur Konfiguration für das *DataTable* (Abb. 8) wird in Listing 17 aufgezeigt.

```
Listing 17:
siehe /app/views/yui_data_table_json.tpl

[...]
var myColumnDefs = [
 {key:"anrede", label:"Anrede", sortable:true },
 {key:"vorname", label:"Vorname" },
 {key:"nachname", label:"Nachname", sortable:true },
 {key:"email", label:"E-Mail", formatter:YAHOO.widget.DataTable.formatEmail },
 {key:"geburtstag", label:"Geburtstag",
       formatter:YAHOO.widget.DataTable.formatDate, sortable:true }
];

var sDataJSON = {{$sDataJSON}};

this.myDataSource =
           new YAHOO.util.DataSource(sDataJSON);
this.myDataSource.responseType = YAHOO.util.DataSource.TYPE_JSON;
this.myDataSource.responseSchema = {
```

```
    resultsList: "ResultSet.person",
    fields: ["titel","anrede","vorname",
             "nachname","geburtstag","telefon",
             "mobiltelefon","email"]
};
[...]
```

Zur Formatierung einzelner Spalten wie beispielsweise *email* können die integrierten *formatter*-Methoden hilfreich sein. Diese sind von Funktion und Ergebnis her ähnlich den *ViewHelper*-Klassen des ZF. Wie bereits weiter oben gezeigt, wird auch hier das Smarty-Tag *{{$sDataJSON}}* im *XmlController* durch den komplett aufbereiteten JSON-Datenblock ersetzt und anschließend die komplett aufbereitete Seite an den Browser geschickt.

Ein bisschen Proxy muss sein

In der zweiten Variante (Abb.7: mit *Datenbeschreibung XML*) werden die XML-Daten mit Hilfe eines PHP-Proxy zum Browser geschickt. Der wesentliche Unterschied zur ersten Variante ist die Reihenfolge wie die Daten zum Browser gelangen. In der ersten Variante wurde Server-seitig die komplette HTML-Seite aufbereitet und zum Browser geschickt. Client-seitig baut YUI aus dem mitgelieferten JSON-Datenblock das oben gezeigte *DataTable* auf. Hierzu erfolgt keine weitere Anfrage zum Server. Der Ablauf in der zweiten Variante ist schon beinahe AJAX, denn nun wird die aufbereitete HTML-Seite ohne Daten zum Browser gesendet. Client-seitig baut YUI das *DataTable* auf und aktiviert dabei den mitgelieferten URL unseres Server-seitigen PHP-Proxy. Dieser liefert auf einen Schwung alle gewünschten XML-Daten zurück zum Client. Das sichtbare Ergebnis beider Varianten unterscheidet sich nicht. Die zweite Variante kann allerdings zu einer echten AJAX-Anwendung ausgebaut werden. Aber auch ohne echtes AJAX müssen noch folgende Punkte geklärt werden, damit die Anwendung sicher ist und innerhalb des MVC Konzeptes läuft.

- Wie kommen die Login-Informationen zum Proxy?
- Wie kann der Aufruf trotz Rewrite funktionieren?

Die erste Frage lässt sich relativ einfach beantworten, alle notwendigen Daten können in einer PHP-Session hinterlegt werden. Hierfür bietet das ZF umfangreiche Funktionalitäten (Listing 18)..

```
Listing 18:
siehe /app/controllers/xmlcontroller.php
[...]

require_once 'Zend/Session/Namespace.php';
$oZF_Session = new Zend_Session_Namespace('getxmldata');
$oZF_Session->df_nPort = $df_nPort;
$oZF_Session->df_sDatabase = $df_sDatabase;
$oZF_Session->df_sOdbcDatabase =
                              $df_sOdbcDatabase;
$oZF_Session->df_sHostname = $df_sHostname;
$oZF_Session->df_sUser = $df_sUser;
$oZF_Session->df_sPassword = $df_sPassword;
$oZF_Session->df_sConnection = $df_sConnection;
$oZF_Session->df_sDatadesc = $df_sDatadesc;

/**
* Die Session ist max. 5 Sekunden gültig.
* Dies sollte ausreichen, damit der PHP-Proxy
* die Abfrage abarbeiten kann.
```

```
*/
$oZF_Session->setExpirationSeconds(5);
$oZF_Session->lock();
[...]
```

Der zweite Punkt ist wirft eine Konzeptfrage auf. Das aktive Rewrite leitet jede Anfrage auf das eingestellte Zielskript (Bootstrap) – für gewöhnlich *index.php* - um. Das bedeutet, auch diese Anfrage vom YUI-DataSource durchläuft den kompletten *FrontController* plus *Dispatcher* plus eventuelle *Plugins* bis hin zum *ProxyController*. Ein gangbarer , aber vermutlich nicht der performanteste Weg. Eine Alternative dazu wäre eine Anpassung der Rewrite-Regeln. Wie das funktionieren könnte zeigt das Szenario beginnend mit Listing 19. Es zeigt einen Teil des aufbereiteten WebSeiten-Quelltext, so wie er zum Browser geschickt wird.

```
Listing 19:
siehe /app/views/yui_data_table_xml.tpl

[...]
var sDataXML = "/proxy/getxmldata?SID=meinesicheresid";

/**
* Aufruf des PHP-proxy, der die komplette
* Ergebnismenge der Abfrage im XML-Format liefert
* (kein Ajax)
*/
this.myDataSource = new YAHOO.util.DataSource(sDataXML);
[...]
```

Das YUI-*DataSource* Objekt führt den Request anhand der Variablen *sDataXML* aus und wartet auf die Daten. Der Request */proxy/getxmldata* wird vom Rewrite auf *index.php* umgeleitet und von dort zum zuständigen Controller weitergeleitet (dispatch). Grundlage hierfür ist folgende Rewrite-Regel, womit alle Anfragen beim Apache-Server - die nicht mit Punkt plus eine der aufgelisteten Erweiterungen enden – auf *index.php,* also auf das Bootstrap-Skript umgeleitet werden.

```
RewriteEngine on
RewriteRule !\.(js|gif|jpg|css)$ /index.php
```

Die Auflistung der Ausnahmeerweiterung ist für gewöhnlich länger, aber für unsere Zwecke reicht sie als Beispiel aus. Wer also einen *ProxyController* nicht mag, der könnte auf die Idee kommen, die *RewriteRule* um den Eintrag |*php*| zu ergänzen. Dies hat zur Konsequenz, dass ein direkter Aufruf eines PHP-Skripts wieder möglich ist, wenn die URL den kompletten Skriptnamen - inklusive der *.php* Erweiterung - enthält. Also auch bei unserem Skript */proxy/getxmldata.php,* das nun unmittelbar ohne den ganzen MVC Überbau ausgeführt wird. Eine einfache und performante Lösung. Kling gut, ist es aber nicht wirklich, denn damit wird jeder direkte Request auf ein PHP-Skript unmittelbar ausgeführt und je nach dem - was da so alles an Testskripten vergessen worden ist – kann dies ein Sicherheitsloch werden.

Ein zweite Möglichkeit ohne einen *ProxyController* auszukommen, bietet eine Ergänzung der Apache Rewrite-Regeln.

```
RewriteEngine on
RewriteCond %{REQUEST_URI} !proxy/getxmldata
RewriteRule !\.(js|gif|jpg|css)$ /index.php

RewriteCond %{REQUEST_URI} proxy/getxmldata
RewriteRule ^/proxy/getxmldata(.*)$  /proxy/getxmldata.php$1
```

Mit Hilfe von *RewriteCond* lässt sich folgender Ablauf realisieren. Zuerst wird geprüft, ob die REQUEST_URI nicht den Suchstring *proxy/getxmldata* enthält, dann wird diese auf *index.php* umgeleitet. Die zweite Regel macht den Umkehrschluss und prüft ob die REQUEST_URI *proxy/getxmldata* enthält und wenn ja, dann wird der Request an */proxy/getxmldata.php* weiter gegeben und zwar inklusive eventueller angehängter GET Parameter wie beispielsweise einer zusätzlichen Security-ID (SID). Ein deutlich sicherer und trotzdem schneller Aufruf des PHP-Proxy. Der PHP-Proxy (Listing 20) liest und prüft als erstes die SID auf dem $_GET Array.

```
Listing 20:
siehe /htdocs/proxy/getxmldata.php

[...]
/**
* die mitgelieferte SID prüfen
*/
$bSidValid = false;

if (array_key_exists('SID',$_GET)) {
  $sSID = (string) $_GET['SID'];
  if ($sSID) {
    // ....SID validieren.....
    $bSidValid = true;
  }
}

if (!$bSidValid) {
 header('Cache-Control: no-cache, must-revalidate');
 header('Content-type: application/xml');
 echo $sXML;
 exit;
}
[...]
```

Ist die SID ungültig, dann wird ein korrekt formatiertes, aber leeres XML-Ergebnis zurück gegeben. Als nächstes werden die Daten aus der *getxmldata* SESSION geholt (Listing 21), siehe dazu auch Listing 18.

```
Listing 21:
siehe /htdocs/proxy/getxmldata.php

[...]
/**
* die SESSION ist max. 5 Sekunden gültig
*/
require_once 'Zend/Session/Namespace.php';
$oZF_Session = new Zend_Session_Namespace('getxmldata');

$df_nPort = $oZF_Session->df_nPort;
$df_sDatabase = $oZF_Session->df_sDatabase;
```

```
$df_sOdbcDatabase = $oZF_Session->df_sOdbcDatabase;
$df_sHostname = $oZF_Session->df_sHostname;
$df_sUser = $oZF_Session->df_sUser;
$df_sPassword = $oZF_Session->df_sPassword;
$df_sConnection = $oZF_Session->df_sConnection;
$df_sDatadesc = $oZF_Session->df_sDatadesc;

$oZF_Session->unsetAll();
[...]
```

Da kann der Proxy in bewährter Methodik die Daten aus der Datenbank lesen und anschließend aufbereitet an den Client und das wartende YUI-DataSource zurück liefern (Listing 22)

```
Listing 22:
siehe /htdocs/proxy/getxmldata.php

[...]
/**
* Eventuelle Zeilenumbrüche entfernen
*/
$sXML = str_replace(chr(13),null,$sXML);
$sXML = str_replace(chr(10),null,$sXML);

/**
* eventuelle Fehlermeldungen protokollieren
*/
if ($sErrorMsg) {
 error_log($sErrorMsg,0);
}

header('Cache-Control: no-cache, must-revalidate');
header('Content-type: application/xml');
echo $sXML;
[...]
```

Abb. 9 zeigt die nun ausgefüllte Tabelle, basierend auf den XML-Daten, die unsere PHP-Proxy an den Browser geschickt hat (Listing 22) und das YUI-*DataTable* im Browser aufbereitet.

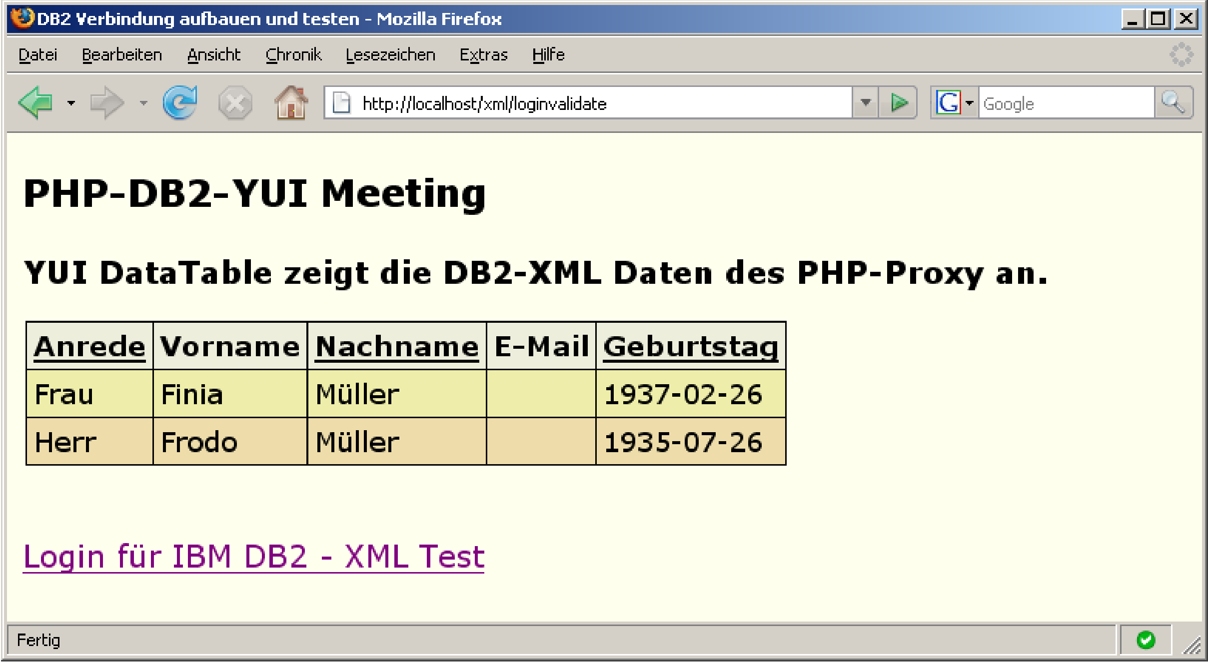


Abb.9: Ergebnistabelle für die XML Abfrage (über den PHP-Proxy)

Dass sich das angezeigte Ergebnis in Abb. 9 vom dem in Abb.8 unterscheidet ist nicht weiter erstaunlich, sondern hängt von der SQL-Abfrage im PHP-Proxy ab. Denn dort werden diesmal nur die Personen abgefragt, die vor dem 01.01.1950 geboren sind.

Zend_View_Helper, Smarty und CSS-Formulare

Abschließend möchte ich noch auf die Möglichkeiten leicht bedienbarer CSS-basierender Formulare zu sprechen kommen. In den Abb. 4 und 7 konnten Sie schon das Ergebnis betrachten. Webdesign und CSS-Formulare waren schon häufiger ein Thema im PHP-Magazin [13], deshalb ist nicht CSS, sondern das Zusammenspiel von *Zend_View_Helper* und *Smarty* im Zentrum des Interesses. Schauen Sie noch einmal auf Listing 5, dort ist zu sehen, wie innerhalb eines Controllers, das Smarty-Objekt *$this->oView* instanziiert wird, mit Hilfe dessen wir Zugriff auf die *View_Helper* erhalten. Das ZF liefert bereits eine Reihe von *View_Helper-Klassen*, für Formularelemente wie *Select, Label, CheckBox, Input* etc. Alle diese erzeugen gültige XHTML-Elemente, die direkt von Smarty genutzt werden können (Listing 23).

```
Listing 23:
siehe /app/controllers/XmlController.php

[...]
/**
* ZF Labelfeldklasse instanziieren
*/
$oZF_FormLabel = $this->oView->getHelper('FormLabel');

/**
* ZF Eingabefeldklasse instanziieren
*/
$oZF_FormText = $this->oView->getHelper('FormText');

/**
* Passwortfeldklasse instanziieren
*/
$oZF_FormPassword = $this->oView->getHelper('FormPassword');
[...]
```

Mit diesen ZF_Form-Objekten können nun die *Smarty-Tags* der Templates gefüllt werden. Schauen wir uns den Eingabeblock des CSS-Formulars im Template (Listing 24) an.

```
Listing 24:
siehe /app/views/xmllogin.tpl

[...]
<form id="login" name="login"
      method="post" action="{{$Action}}" accept-charset="ISO-8859-1">
 <fieldset>
   <legend>Login für IBM DB2 - XML Test</legend>
   <div class="datafield" id="Hostname" >
     {{$labelHostname}}{{$df_sHostname}}
     <span class="error">{{$error_df_sHostname}}
     </span>
   </div>
[...]
```

Alle Eingaben werden zu einem *fieldset* zusammen gefasst. Jeden einzelnen Eingabeblock umschließt ein *div* mit eindeutiger *id*. Die *Smarty-Tags* erkennen Sie wieder an den doppelten geschweiften Klammern. Beim *rendern* ersetzt *Smarty* diese Platzhalter durch die von den *View_Helpern* - zur Laufzeit erzeugten - XHTML-Elementen (Listing 25).

```
Listing 25:
siehe /app/controllers/XmlController.php

[...]
/**
* Label für Eingabefelder erstellen, die Zuordnung
* erfolgt über die jeweilige
* z.B. id="df_sHostname" im Sinne von:
* "Datenfeld (df_)-String (s)-Feldname (Hostname)"
*/
$labelHostname = $oZF_FormLabel->formLabel('df_sHostname','Hostname:',array());
[...]

/**
* Eingabefelder erstellen
*/
$df_sHostname = $oZF_FormText->formText('df_sHostname', $df_sHostname,
    array( 'tabindex' => '1',  'size' => '15', 'maxlength' => '15'));
[...]

/**
* Auswahlliste
*/
$select_Connection = $oZF_FormSelect->formSelect('df_sConnection',
   $df_sConnection, array( 'tabindex' => '7', 'size' => '1'),
    array('IBM_DB2' => 'IBM_DB2', 'PDO_IBM' => 'PDO_IBM'));
[...]
```

Nach diesem Konzept werden alle *Smarty-Tags* des Templates ausgefüllt. Neben jeder Eingabe gibt es einen <*span*>-Bereich für eine eventuelle Fehlermeldung nach der Validierung. Die Überprüfung der Feldinhalte erfolgt Server-seitig. Bleiben wir aber noch kurz bei diesem Formular, denn abschließend werden alle aufbereiteten XHTML-Elemente mittels eines Array an Smarty übergeben (Listing 26).

```
Listing 26:
siehe /app/controllers/XmlController.php

[...]
/**
* Label und Eingabefeld dem Smarty-Template als
* array() zuweisen
*/
$aFields = array(
  'labelHostname' => $labelHostname,
  'df_sHostname' => $df_sHostname,
  'labelDatabase' => $labelDatabase,
  'df_sDatabase' => $df_sDatabase,
  'labelOdbcDatabase' => $labelOdbcDatabase,
  'df_sOdbcDatabase' => $df_sOdbcDatabase,
  'labelPort' => $labelPort,
  'df_nPort' => $df_nPort,
  'labelUser' => $labelUser,
  'df_sUser' => $df_sUser,
  'labelPassword' => $labelPassword,
  'df_sPassword' => $df_sPassword,
  'labelConnection' => $labelConnection,
```

```
  'select_Connection' => $select_Connection,
  'labelDatadesc' => $labelDatadesc,
  'select_Datadesc' => $select_Datadesc,
  'labelLogin' => $labelLogin,
  'pb_Login' => $buttonSubmit );

$this->oView->assign($aFields);


/**
* Sind Validierung-Fehlermeldungen vorhanden,
* dann diese in den vorgesehenden
* <span class="error">{{$error_xxxxx}}</span>
* ausgeben.
*/
if ($aErrorMsg) {
 foreach($aErrorMsg as $key => $value) {
  $this->oView->assign('error_'.$key,$value);
 }
}

/**
* Smarty Template ausfüllen und das Ergebnis
* an den Browser schicken
*/
echo $this->oView->render('xmllogin.tpl');
[...]
```

Mit dem Befehl *echo $this->oView->render('xmllogin.tpl')* wird das Template aufbereitet und der Inhalt an den Browser gesendet.

Resumee

Mit einer Reihe von Hinweisen, Tipps und Tricks habe ich Ihnen gezeigt, wie Sie wichtige Komponenten und Tools zu einem Web-Projekt zusammenführen können. Ziel war es, möglichst praxisnah auf die wesentlichen Schnittstellen von *PDO, DB2, XML, Zend Framework, Smarty* und *YUI* hinzuweisen. Zu jeder der hier aufgezeigten Lösungen gibt es selbstverständlich alternative Wege und Varianten. Im ZF-, YUI- und PHP-Umfeld werden ständig Weiterentwicklungen, neue Ideen und *best practices* vorgestellt, trotzdem ist es häufig wichtig zu wissen: Ob und wie etwas überhaupt realisiert werden kann. Das *Zend Framework* dient dabei als Rahmen meiner Problemlösung und wird auch gerade deshalb gelobt, weil es dem Entwickler die Möglichkeit lässt, seine eigenen Konzepte und Ideen zu verwirklichen.

Thomas Wiedmann ist Anwendungs- und Datenbankentwickler für individuelle Softwarelösungen, Autor des Buches DB2, Zend Certified Engineer, sowie IBM Certified DB2 Administrator. Wenn er nicht gerade Musik hört, liest oder mit dem Rad unterwegs ist, beschäftigt er sich mit Datenbanken und Web-Applikationen.

Links & Literatur

[1] PHP Download http://www.php.net PECL http://pecl.php.net
[2] Zend Framework http://framework.zend.com
[3] Smarty Template http://smarty.php.net
[4] YUI http://developer.yahoo.com/yui/
[5] IBM DB2 Express-C Download http://www.ibm.com/software/data/db2/udb/db2express/
[6] Apache HTTP Server http://www.apache.org

[7] IBM DB2 Express-C Installation
http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0602hutchison/
http://www.ibm.com/developerworks/wikis/display/DB2/FREE+Book-+Getting+Started+with+DB2+Express-C
[8] Carsten Möhrke, Zend Framework – Das Entwickler-Handbuch. Galileo Press 2008
[9] Firebug Add-On http://www.getfirebug.com
[10] Yahoo! Graded Browser Support http://developer.yahoo.com/yui/articles/gbs
[11] Zend Core for DB2 http://www.zend.com/de/products/core/for-ibm
[12] DB2 9 pureXML Guide Redbook SG24-7315-01 http://ibm.com/redbooks
[13] Stefan Blanz, Bedienbare Formulare. PHP Magazin 05.2006, Seite 92ff.

Eine Haftung für die Richtigkeit der Veröffentlichungen kann trotz sorgfältiger Prüfung durch den Herausgeber nicht übernommen werden. Kein Teil dieser Publikation darf ohne ausdrückliche schriftliche Genehmigung des Herausgebers in irgendeiner Form reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden. Die Nutzung der Programme ist nur zum Zweck der Fortbildung und zum persönlichen Gebrauch des Lesers gestattet.