Ez publish 3.* Tutorial

Einführung in die Entwicklung

von eZ publish Extensions

Dipl.-Ing. Felix Woldt (FH)

felix_[at]_jac-systeme.de

www.jac-systeme.de

Stand April 2007

Inhalt

1 Einführung in die Entwicklung von eZ publish Extensions

1.1 Einleitung

Viele Anforderungen an ein Redaktionssystem können mit eZ publish ohne PHP Programmierung bereits abgebildet werden. Früher oder später kommt jeder Entwickler aber an den Punkt, wo es spezielle Anforderungen in einem Projekt erfordern, eigene Erweiterungen (Extensions) zu entwickeln. Dieser Artikel soll den Einstieg an Hand eines einfachen Beispiels in diese sehr interessante Materie erleichtern.

1.1.1 Was sollten Sie wissen/können...

- Installation von eZ publish und Grundverständnis über den Aufbau von eZ publish
- Grundkenntnisse in PHP, SQL, MYSQL, HTML und objektorientierter Programmierung sind von Nutzen

1.1.2 Was wird gezeigt

In diesem Artikel erfahren Sie, wie eine einfache Extension angelegt und konfiguriert wird. Dabei wird gezeigt, wie das eZ publish Framework bei der Entwicklung genutzt werden kann.

1.2 Was sind Extensions

Bevor wir mit dem Beispiel starten, möchte ich kurz den Begriff Extension erläutern und auf den Aufbau von Extensions eingehen.

Eine Extension erweitert die vorhandene Funktionalität von eZ publish, ohne die originalen Dateien zu verändern. Die Trennung der neuen Funktionalität in separate Extensions hat den Vorteil, dass spätere Updates auf eine neue eZ publish Version einfach möglich sind.

Durch Extensions ist es möglich z.B.

- das Design einer Webseite / 'siteaccess' in eine Extension auszulagern,
- eigene Module mit neuen 'views' und Template 'fetch' Funktionen anzulegen,
- das Templatesystem mit eigenen Template Operatoren zu erweitern,
- neue Workflow events , Datentypen oder Login Handler zu programmieren,

Dies ist nur eine kleine Auswahl an Möglichkeiten.

Extensions in eZ publish haben meist einen ähnlichen Aufbau (siehe Tabelle 1).

Extension Unter-verzeichnisse	**Beschreibung**
actions	neue Aktionen für Formulare
autoloads	Definitionen von eigenen Template Operatoren
datatypes	Definitionen für neue Datentypen
design	Dateien (*.tpl, *.css, *.jpg, *.js ...) die etwas mit dem Design zu tun haben.
eventtypes	benutzerdefinierte Workflow events
modules	ein oder mehrere Module mit Views, Template Fetch Funktionen ...
settings	Konfigurtiondateien (*.ini, *.ini.append.php) der Extension
translations	Übersetzungsdateien (*.ts)

Tabelle 1: Mögliche Verzeichnisstruktur einer eZ publish 3.x Extension

Die dargestellte Verzeichnisstruktur in Tabelle 1 ist nur ein Beispiel und ist abhängig von der Art der Extension. Es sind nicht immer alle Ordner nötig z.B. bei einer Template Operator Extension wird nur der Ordner *autoloads* und *settings*, bei einer Modul-Extension hingegen der Ordner *modules* und *settings* und eventuell *design* benötigt.

1.3 Neue Extension am Beispiel von jacextension

Am Beispiel der Extension *jacextension* soll der Einstieg in die grundlegenden Dinge der Entwicklung von eZ publish Extensions gegeben werden. Es werden alle notwendigen Schritte für das Anlegen einer neuen Extension dargestellt und die gängigsten PHP Klassen des eZ publish Framework vorgestellt. Folgende Sachverhalte werden besprochen:

- Zugriff auf die Datenbank (eZ Framework- Klasse eZPersistentObject)
- Zugriff auf .ini Konfigurationsdateien (eZ Framwork - Klasse eZINI)
- Zugriff auf GET, POST SESSION Variablen (eZ Framework - Klasse eZHTTPTOOL)
- Erzeugen eigener Debug Meldungen / Logdateien (eZ Framework - Klasse eZDebug)
- Nutzung des eZ Rechtesystems für neue Views eines Moduls
- Erweitern des Templatesystems durch eigene Template Fetch Funktionen bzw. Operatoren

1.3.1 Voraussetzungen

Um mit dem Beispiel beginnen zu können, muss zuvor eZ publish erfolgreich installiert werden. Dazu richten wir uns über den eZ Setup Assistenten die *Plain Site*, im Access Modus *URL*, auf der MySql Datenbank *ez39_plain* (Zeichensatz utf-8) ein. Dadurch werden zwei Siteaccesse *plain_site* und *plain_site_admin* angelegt. Folgende URL Aufrufe sollten dann funktionieren:

http://localhost/ez/index.php/plain_site (Benutzeransicht)

http://localhost/ez/index.php/plain_site_admin (Administrationsansicht)

Dabei verweist *localhost/ez/* direkt auf das eZ publish Installtionsverzeichnis.

Einrichtung Extension

Wir wollen nun eine neue Extension mit dem Namen *jacextension* anlegen und diese aktivieren. Diese soll für den Anfang ein Modul namens *modul1* mit einer View *list* beinhalten, die das PHP Script *list.php ausführt.*

Dazu wechseln wir in den Ordner *extension* des eZ publish Root Ordners und legen dort einen neuen Ordner *jacextension* mit den Unterordnern und PHP Dateien - wie nachfolgend dargestellt - an:

Ezroot/ extension /

+--jacextension

+-- modules

+-- modul1

+-- list.php

+-- module.php

+--settings

+--module.ini.append.php

Mit der Konfigurationsdatei *module.ini.append.php* (Listing 1) teilen wir dem eZ publish System mit, dass es in der Extension *jacextension* nach Modulen suchen soll. Daraufhin versucht eZ publish alle Module wie z.B. *modul1* , die sich im Verzeichnis *extension/jacextension/modules/* befinden, zu laden.

Tipp: Bei INI Einträgen darauf achten, dass keine überflüssigen Leerzeichen am Zeilenende (vor dem Umbruch) stehen, da sonst die Werte nicht richtig interpretiert werden.

1.3.2 View list

In der *module.php* werden alle Views eines Moduls definiert (siehe Listing 2). Über eine View kann auf eine PHP Datei zugegriffen werden. In unserem Beispiel definieren wir mit $ViewList['list'] den Namen der View *list* und leiten auf die PHP Datei *list.php* weiter. Wir

wollen der View auch noch Parameter übergeben. Beispielhaft definieren wir uns in der Datei *module.php* zwei Parameter *ParamOne* und *ParamTwo* mit 'params' => array('ParamOne', 'ParamTwo') . Der zugehörige URL Aufruf sieht so aus: *http://localhost/ez/index.php/plain_site/list/ValueParamOne/ValueParamTwo/*. Auf die Variablen kann später in der *list.php* über folgendende Aufrufe zugegriffen werden: $valueParamOne = $Params['ParamOne']; und $valueParamTwo = $Params['ParamTwo']. Die dargestellte Form des Parameterzugriffes wird auch als *ordered parameters* bezeichnet. Daneben gibt es noch *unordered parameters*. Diesen werden den *ordered parameters* nachgestellt und bestehen immer aus einem Namen-Werte-Paar z.B. *...modul1/list/ValueParamOne/ValueParamTwo/ NameParam3/ValueParam3/ NameParam4/ValueParam4* . Die Definition erfolgt ähnlich wie bei den *ordered parameters* in der *module.php* mit 'unordered_params' => array('param3' => '3Param', 'param4' => '4Param'). Wenn der Viewaufruf z.B. in der URL den Wert .../parm4/141 dann erkennt eZ das der Parameter param4 gesetzt ist und wir können den Wert 141 aus dem Parameter Array des Moduls über $Params['4Param'] auslesen. Die *unordered parameter* dürfen in der Reihenfolge vertauscht werden und können auch ungesetzt bleiben. Wenn ein *unordered parameter* nicht angegeben wird, dann setzt eZ den Wert auf false. Im Vergleich dazu müssen die *ordered parameter* angegeben werden. Ansonsten erhalten die Wert NULL. Folgender URL Aufruf *http://localhost/ez/index.php/plain_site/modul1/list/table/5/param4/141/param3/131* würde demnach die View Parameter wie folgt setzen:

$Params['ParamOne'] = 'table';

$Params['ParamTwo'] = '5';

$Params['3Param'] = '131';

$Params['4Param'] = '141';

Siehe dazu auch die eZ Dokumentation unter

http://ez.no/doc/ez_publish/technical_manual/3_8/concepts_and_basics/modules_and_views.

Es ist üblich in eZ publish für jede View eine PHP Datei mit gleichem Namen anzulegen. So kann anhand des URL Aufrufs leicht erkannt werden, welche PHP Datei aufgerufen wird z.B:

http://localhost/ez/index.php/plain_site/content/view/full/2

leitet auf die Datei *view.php* im Kernelmodul *content* weiter (*ezroot/kernel/content/view.php*) .

Kleiner Tipp: An den eZ Kernel Modulen kann man eine Menge lernen, also ruhig einmal den Quellcode anschauen. Der Aufbau ist identisch wie ein Modul aus einer Extension, wie z.B. bei unserem Beispielmodul *modul1*.

Damit wir beim Aufruf der View *list* auch eine Ausgabe auf dem Bildchirm sehen, schreiben wir noch eine echo Anweisung in die Datei *list.php* , die die übergebenen Parameter auf dem Bildschirm ausgibt (siehe Listung 3).

1.3.3 Aktivierung Extension

Um die neu erstellte View *list* des Moduls *modul1* in der Extension *jacextension* testen zu können, müssen wir noch die Extension aktivieren. Dies erfolgt entweder in der globalen *site.ini.append.php* (siehe Listing 4) oder in der *site.ini.append.php* des jeweiligen Siteaccess (siehe Listing 5). Dabei unterscheiden sich die Aktivierungschreibweisen ActiveExtensions[] und ActiveAccessExtensions[]. Die globale Aktivierung gilt dabei für alle Siteaccesse der eZ Installation. In unserem Beispiel *plain_site* und *plain_site_admin*.

1.3.4 Rechtesystem

Wenn wir jetzt versuchen die View *list* des Moduls *modul1* über die URL

http://localhost/ez/index.php/plain_site/modul1/list/table/5/param4/141

aufzurufen, bekommen wir eine Meldung von eZ publish 'Zugriff verweigert'. Warum? Der Benutzer *Anonymous* hat nicht die nötige Berechtigung, die View *list* des Moduls *modul1*

aufzurufen. Um die nötigen Rechte zu erteilen, gibt es zwei Möglichkeiten. Erstens können wir den Zugriff explizit für alle Benutzer über den Eintrag PolicyOmitList[]=modul1/list in der neu angelegten Konfigurationsdatei e*xtension/jacextension/settings/site.ini.append.php,* erlauben (siehe Listing 6). Oder aber wir steuern den Zugriff über das Benutzer- und Rollensystem von eZ publish. Dazu müssen wir die Rolle *Anonymous* so abändern, dass wir den Zugriff auf die entsprechenden Funktionen des Moduls *modul1* gestatten. Dies kann unter folgender URL erfolgen: *http://localhost/ez/index.php/plain_site_admin/role/view/1* . Welche Funktionen ein Extensionmodul zur Auswahl stehen, wird in der *module.php* über das Array $FunctionList definiert. Die Verknüpfung erfolgt in der Definition der View mit dem Arraykey *functions*. So verknüpfen wir in unserem Beispiel die View *list* mit der Rollenfunktion *read* ($FunctionList['read'] = array()) mit 'functions' => array('read'). Über die $FunctionList ist es möglich einzelne Views eines Moduls mit bestimmten Rollenfunktionen zu verknüpfen. So hat man z.B. noch eine weitere Rollenfunktion *create* , die allen Views zugeordnet wird, die Inhalte anlegen. Zugriff auf diese könnten wir z.B. nur Redakteuren geben. Die Rollenfunktion *read* verknüpfen wir mit allen Benutzern/Rollen die Leserechte bekommen sollen, so z.B. auch mit dem Benutzer *Anonymous*.

Listing 1. Modul-Konfigurationsdatei: extension/jacextension/settings/module.ini.append.php

```
<?php /* #?ini charset="utf-8"?
# eZ mitteilen, dass in jacextension nach Modulen gesucht werden soll
[ModuleSettings]
ExtensionRepositories[]=jacextension
*/ ?>
```

Listing 2. View-Konfigurationsdatei: extension/jacextension/modul1/module.php

```
<?php
$Module = array( 'name' => 'Example Modul1' );
$ViewList = array();
// neue View list mit 2 festen Parametern und
// 2 in der Reihenfolge variablen Parametern
//   http://.../modul1/list/   $Params['ParamOne']   /   $Params['ParamTwo']/
param4/$Params['4Param'] /param3/$Params['3Param']
```

```
$ViewList['list'] = array( 'script' => 'list.php',
                           'functions' => array( 'read' ),
                           'params' => array('ParamOne', 'ParamTwo'),
                           'unordered_params' =>
                    array('param3' => '3Param', 'param4' => '4Param') );

// Die Funktionlisteinträge (Rollenfunktionen) werden
// in der Viewdefintion genutzt, damit in den Benutzerrollen auf
// einzelne View Funktionen Rechte vergeben werden können

$FunctionList = array();
$FunctionList['read'] = array();
?>
```

Listing 3. Funktionsdatei der View list: *extension/jacextension/modul1/list.php*

```
<?php
// aktuelle Objekt vom Typ eZModule holen
$Module =& $Params['Module'];
// Ordered View Paremeter auslesen
// http://.../modul1/list/ $Params['ParamOne'] / $Params['ParamTwo']
// z.B    .../modul1/list/view/5
$valueParamOne = $Params['ParamOne'];
$valueParamTwo = $Params['ParamTwo'];

// UnOrdered View Paremeter auslesen
//http://.../modul1/list/param4/$Params['4Param']/param3/$Params['3Param']
// z.B    .../modul1/list/.../.../param4/141/param3/131
$valueParam3 = $Params['3Param'];
$valueParam4 = $Params['4Param'];

// Werte der View Parameter ausgeben
echo 'Example: modul1/list/'
              .$valueParamOne .'/ '. $valueParamTwo.'/param4/'
              .$valueParam4.'/ param3/'.  $valueParam3;
?>
```

Listing 4. Möglichkeit 1 - Extensionaktivierung für alle verfügbaren Siteaccesse in globaler Konfigurationsdatei: *settings/override/site.ini.append.php*

```
<?php /* #?ini charset="utf-8"?
[ExtensionSettings]
ActiveExtensions[]
ActiveExtensions[]=jacextension
...
*/ ?>
```

Listing 5. Möglichkeit 2 - Extensionaktivierung in Konfigurationsdatei eines Siteaccesses z.B. *settings/ siteacces/ plain_site/ site.ini.append.php*

```
<?php /* #?ini charset="utf-8"?
...
[ExtensionSettings]
ActiveAccessExtensions[]=jacextension
...
*/ ?>
```

Listing 6. Zugriffsreche auf die View *list* des Moduls *modul1* für alle eZ Benutzer setzen, in Konfigurationsdatei: *extension/jacextension/settings/site.ini.append.php*

```
<?php /* #?ini charset="utf-8"?
[RoleSettings]
PolicyOmitList[]=modul1/list
*/ ?>
```

1.3.5 Templatesystem

Da uns die *echo* Ausgabe des PHP Scripts *list.php* nicht befriedigt, möchten wir jetzt ein eigenes Template nutzen. Dazu legen wir die list.tpl in den Ordner *jacextension/design/standard/templates/modul1/list.tpl* . Damit eZ publish das Template später auch finden kann, müssen wir die *jacextension* noch als Designextension deklarieren. Dazu legen wir die Konfigurationsdatei *design.ini.append.php* im Ordner *.../jacextension/settings/* an (siehe Listing 7).

In *list.php* erzeugen wir uns dann eine TemplateInstance $tpl =& templateInit(); . Anschließend setzen das View Parameter Array $viewParameters und das Array mit den Beispieldaten $dataArray als Templatevariable {$view_parameters} beziehungsweise {$data_array} mit der Anweisung $tpl->setVariable('view_parameters', $viewParameters); und $tpl->setVariable('data_array', $dataArray);. Dann Parsen wir das Template *list.tpl* mit den gesetzten Variablen. In unserem Fall nur $view_parameters und $dataArray und speichern das Ergebnis in $Result['content']. In dem eZ Haupttemplate *pagelayout.tpl* kann dann über die Variable {$module_result.content} das Ergebnis ausgelesen werden, was standardmäßig gemacht wird. Zum Abschluss setzen wir noch den Pfad *Modul1/list* der im Browser angezeigt werden soll. In unserem Beispiel kann man auf den 1. Teil des Pfades klicken, der auf *modul1/*list verweist. (siehe Listing 8)

Nun können wir im Template *list.tpl* auf die gesetzten Variablen zugreifen mit {$view_parameters} und {$data_array}. Die übergebenen View Parameter lassen wir uns mit {$view_parameters|attribute(show)} anzeigen. Dann prüfen wir mit dem Template Operator is_set($data_array) ob die Variable $data_array existiert und geben entweder eine Liste mit den Daten beziehungsweise eine Standard Nachricht aus (siehe Listung 9).

Wenn wir nun unsere View über *http://localhost/ez/index.php/plain_site/modul1/list/table/5* aufrufen, passiert nicht wirklich viel. Es erscheint nur der Pfad *Modul1/list*. Warum? Wir wissen es nicht! Um dem Fehler auf die Spur zu kommen, aktivieren wir die eZ Debug Ausgabe inklusive der zurzeit genutzten Templates. Das Compilieren und Cachen der Templates schalten wir auch noch aus, um ganz sicher zu gehen, dass alle Templateänderungen angezeigt werden. Dazu erweitern wir die globale *ezroot/settings/override/site.ini.append.php* um entsprechende Einträge (siehe Listing 10). Rufen wir jetzt *.../modul1/list/table/5* erneut auf, sollte in der Debug Ausgabe folgende Template Warnung stehen: 'No template could be loaded for "modul1/list.tpl" using resource "design"' . Das Template *list.tpl* wird anscheinend nicht gefunden. In diesem Fall hilft es,

einmal den kompletten eZ Cache zu löschen, da eZ publish die Liste der bis dahin verfügbaren Templates gecached hat. Dazu die URL *http://localhost/ez/index.php/plain_site_admin/setup/cache* aufrufen und auf den Button 'Den kompletten Cache leeren' klicken. Nun sollte das *list.tpl* Template mit der Tabellenansicht der View Parameter und unserer Beispieldatenliste angezeigt werden. Weiterhin muss ein Eintrag *modul/list.tpl* in der Debug Ausgabe 'Templates used to render the page:' vorhanden sein.
In unserem Beispiel haben die View Parameter folgende Werte $view_parameters.param_one = 'table' und $view_parameters.param_one = '5' . Diese können im PHP Script *list.php* oder im Template *list.tpl* genutzt werden, um Zustände zu steuern z.B. wie die Ausgabe dargestellt wird oder welcher Datensatz mit einer bestimmten ID angezeigt werden soll.
Tipp: die Templatevariable $view_parameters ist unter anderem auch im eZ Kernel Modul *content* verfügbar, also in den meisten Templates, wie z.B. *node/view/full.tpl.*

Listing 7. Extension jacextension als Designextension deklarieren

```
<?php /* #?ini charset="utf-8"?
# eZ Mitteilen, in jacextension nach designs zu suchen
[ExtensionSettings]
DesignExtensions[]=jacextension
*/ ?>
```

Listing 8. modul1/list.php – Erweiterung von Listing 3

```
<?php
// Bibliothek für Templatefunktionen
include_once( "kernel/common/template.php" );
...
// Templateobject initialisieren
$tpl =& templateInit();

// View Parameter Array zum Setzen ins Template erzeugen
```

```
$viewParameters = array( 'param_one' => $valueParamOne,
                         'param_two' => $valueParamTwo,
                         'unordered_param3' =>  $valueParam3,
                         'unordered_param4' =>  $valueParam4);

// Parameter der View als Array dem Template übergeben
$tpl->setVariable( 'view_parameters', $viewParameters );
// Beispieldatenarray im Template setzen => {$data_array}
$tpl->setVariable( 'data_array', $dataArray );
...
// Template parsen und Ergebnis für $module_result.content speichern
$Result['content'] =& $tpl->fetch( 'design:modul1/list.tpl' );
...
?>
```

Listing 9. eZ Template design:modul/list.tpl

```
{*   list.tpl  -  Template  für  die  Modulview  .../modul1/list  /  ParamOne  /
ParamTwo

    Prüfen ob Variable $data_array vorhanden ist
  - ja:   Daten als Liste ausgeben
  - nein: Meldung ausgeben
*}
Array $view_parameters anzeigen:
{$view_parameters|attribute(show)}<br />

<h1>Template: modul1/list.tpl</h1>
{if is_set($data_array)}
<ul>
  {foreach $data_array as $index => $item}
  <li>{$index}: {$item}</li>
  {/foreach}
</ul>
{else}
<p>Achtung: keine Daten vorhanden!!</p>
{/if}
```

Listing 10. Debug Ausgabe mit Templateliste einschalten über globale Konfigurationsdatei:

ezroot/ settings/override/site.ini.append.php

```
<?php /* #?ini charset="utf-8"?
...
[DebugSettings]
Debug=inline
DebugOutput=enabled
DebugRedirection=disabled

[TemplateSettings]
ShowXHTMLCode=disabled
ShowUsedTemplates=enabled
TemplateCompile=disabled
TemplateCache=disabled
*/ ?>
```

1.3.6 View create

Jetzt erweitern wir unser Beispiel um eine neue View *create*. Wir wollen nun das Array mit den Beispieldaten in einer eigenen Datenbanktabelle abspeichern. Dazu legen wir uns eine neue Datenbanktabelle mit dem Namen *jacextension_data* in unserer eZ Datenbank *ez39_plain* mit den Spalten *id* | *user_id* | *created* | *value* an, z.B. mit PhpMyAdmin (siehe Listing 11).

Listing 11. Folgende SQL Anweisung in der eZ Datenbank ez39_plain ausführen, um ein neue Tabelle *jacextension_data* anzulegen.

```
CREATE TABLE `jacextension_data` (
`id` INT( 11 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`user_id` INT( 11 ) NOT NULL ,
`created` INT( 11 ) NOT NULL ,
`value` VARCHAR( 50 ) NOT NULL
) ENGINE = MYISAM ;
```

Dann kopieren wir *list.php* nach *create.php*, *list.tpl* nach *create.tpl* und erweitern die *module.php* um eine neue View und Rollenfunktion *create* die auf die PHP Datei *create.php* weiterleitet. Dann ändern wir nur noch den Templateaufruf in *create.php* auf *design:modul1/create.tpl* um und passen die Rechte für die View *modul1/create* entsprechend an. Den Cache noch schnell löschen. Nun sollte der URL Aufruf *http://localhost/ez/index.php/plain_site/modul1/create* funktionieren (selbe Ausgabe wie *modul1/list* nur ohne View Parameter).

Damit wir einen neuen Datensatz in der MySql Datenbank speichern können, brauchen wir ein HTML Formular, dass die Daten an unsere neue View *create* schickt (über POST oder GET Variablen). Deshalb legen wir im Template *create.tpl* ein neues Formular mit einer Texteingabezeile *name* an. In unserem Beispiel wollen wir die Daten per GET übertragen. Wir fügen auch noch gleich eine Templatevariable {$status_message} ein, damit wir hier eine Nachricht für den Benutzer ausgeben können (siehe Listing 12).

Listing 12. Template jacextension/design/standard/templates/modul1/create.tpl mit einem Formular zum Erfassen der neuen Daten

```
{*  create.tpl - Template für die Modulview .../modul1/create
              Htmlformular für das Erfassen neuer Datensätze *}
<form action={'modul1/create'|ezurl()} method="get">
  Name:<br />
  <input name="name" type="text" size="50" maxlength="50"><br />
  <input type="submit" value="Neuen Datensatz anlegen">
  <input type="reset" value="Abbrechen">
</form>

<hr>
Status: {$status_message}
```

1.3.7 GET / POST

Anschließend ändern wir die *list.php* so ab, dass wir die GET Variable *name*, die vom HTML Formular an das Script übergeben wird, ausgelesen wird. Dann geben wir diese im Statusfeld des Templates aus. Zum Auslesen der GET / POST Variablen nutzen wir das eZ publish Framework mit der Klasse eZHTTPTool. Zuerst holen wir uns die globale Objektinstanz von eZHTTPTool mit $http =& eZHTTPTool::instance(); . Mit $http->hasVariable('name'); können wir herausfinden, ob die Variable $_GET['name'] oder $_POST['name'] existiert und mit $http->variable('name'); können wir diese auslesen. Möchte man nur GET oder POST Variablen zulassen, kann dies durch $http->hasGETVariable('name'); oder $http->hasPOSTVariable('name'); erfolgen. Welche Funktionen es noch gibt, wie z.B. auf SESSIONS zugegriffen werden kann, kann in der API Dokumentation von eZ publish nachgelesen werden. Für die Klasse eZHTTPTool ist diese zu finden unter folgender URL: *http://pubsvn.ez.no/doxygen/classeZHTTPTool.html* .

1.3.8 Debug

Dann setzen wir unsere Templatevariable {$status_message} mit dem übergebenen Formularwert oder einem Standardtext. Über eZDebug::writeDebug() kann das Ergebnis oder eine andere Meldung in die eZ publish Debug Ausgabe und in die entsprechende Log Datei z.B. *ezroot/var/log/debug.log* geschrieben werden. Weitere Anweisungen zur Ausgabe von Informationen sind z.B. writeError() oder writeNotice(). Unter der API Dokumentation zu eZDebug kann man alles genauer nachlesen: *http://pubsvn.ez.no/doxygen/classeZDebug.html* . Wenn wir eine eigene Logdatei schreiben möchten, geht das über eZLog::write(). Das ist manchmal ganz nützlich, da die Standardlogfiles viele verschiedene Informationen enthalten können; und das ist manchmal ziemlich unübersichtlich (siehe Listing 13).

Listing 13. jacextension/module/modul1/create.php mit Beispielen zum Auslesen von GET/POST Variablen und generieren von Debugmeldungen und eigener Log Dateien

```
<?php
// modul1/create.php - Funktionsdatei der View create
// notwendige php Bibliothek für Templatefunktionen bekannt machen
include_once( "kernel/common/template.php" );

$Module =& $Params['Module'];
// globle objectinstanz holen
$http =& eZHTTPTool::instance();
$value = '';

// Wenn Variable 'name' per GET oder POST übertragen wurde, Variable auslesen
if( $http->hasVariable('name') )
    $value =  $http->variable('name');

if( $value != '' )
    $statusMessage = 'Name: '. $value;
else
    $statusMessage = 'Bitte Daten eingeben';

// Templateobject initialisieren
$tpl =& templateInit();
$tpl->setVariable( 'status_message', $statusMessage );

// Variable $statusMessage in den eZ Debug Output / Logdatei schreiben
// hier die 4 verschiedenen Typen Notice, Debug, Warning, Error
eZDebug::writeNotice( $statusMessage, 'jacextension:modul1/list.php');
eZDebug::writeDebug( $statusMessage, 'jacextension:modul1/list.php');
eZDebug::writeWarning( $statusMessage, 'jacextension:modul1/list.php');
eZDebug::writeError( $statusMessage, 'jacextension:modul1/list.php');

// $statusMessage eigene Logdatei schreiben nach ezroot/var/log/jacextension_modul1.log
```

```
include_once('lib/ezfile/classes/ezlog.php');
eZLog::write( $statusMessage, 'jacextension_modul1.log', 'var/log');

$Result = array();
// Template parsen und Ergebnis für $module_result.content speichern
$Result['content'] =& $tpl->fetch( 'design:modul1/create.tpl' );
// Pfad generieren   Modul1/create
$Result['path']   =   array(   array(   'url'   =>   'modul1/list','text'   =>
'Modul1' ),
                          array( 'url' => false,'text' => 'create' ) );
?>
```

1.3.9 Datenbankzugriff

Nun wollen wir aber auf die Datenbank zurückkommen. Wir möchten jetzt den übergebenen Formularwert in unsere neu angelegte Tabelle *jacextension_data* speichern. Für diesen Zweck gibt es in eZ publish die Klasse eZPersistentObject. Diese beinhaltet schon alle gängigen Funktionen zum Anlegen, Verändern, Löschen oder Extrahieren von Datensätzen. Um die Funktionalität nutzen zu können, erzeugen wir eine neue Klasse JACExtenstionData. Diese speichern wir im Ordner *ezroot/extension/jacextension/classes* mit dem Namen *jacextensiondata.php* ab (siehe Listing 14). Die wichtigste Funktion ist JACExtenstionData::definition(). Hier wird die Struktur des JACExtensionData Objektes definiert und angegeben, in welcher Tabelle mit welchen Tabellenspalten die Daten gespeichert werden sollen. Dann legen wir uns noch die Funktionen create(), fetchByID(), fetchList(), getListCount() an. Es werden drei verschiedene Arten gezeigt, wie Daten ausgelesen werden können (eZPersistentObject::fetchObject(), eZPersistentObject::fetchObjectList(), direkte SQL Anfrage). Wenn es irgendwie möglich ist, sollten die eZPersistentObject fetch Funktionen genutzt werden. Es sollten keine datenbankspezifischen SQL Statements verwendet werden, damit die SQL Abfragen auf den von eZ publish unterstützten Datenbanksystem z.B. Mysql, Postgres, Oracle funktionieren. (siehe auch API *http://pubsvn.ez.no/doxygen/classeZPersistentObject.html*)

Anschließend nutzen wir die neu erstellten Funktionen im Script *create.php.* Um einen neuen Datensatz abzuspeichern, erzeugen wir uns ein neues Objekt vom Typ JACExtensionData über die Funktion JACExtensionData::create($value) . Die Funktion create() setzt dabei gleichzeitig den übergebenen *value* (Wert aus Formularfeld), die aktuelle *user_id* und die aktuelle Zeit *created.* Mit der Funktion store() speichern wir nun den Datensatz in die Datenbanktabelle *jacextension_data*. Um zu sehen wie sich die Daten verändern, werden diese zwischendurch in die Debug Ausgabe geschrieben (siehe Listing 15).

Auf Objekte, die vom Typ eZPersitentObjekt sind, können wir ganz einfach mit $JacDataObject->attribute('id') zugreifen. Dabei entspricht der Funktionsparameter *'id'* der Tabellespalte *id*. So brauchen wir nicht lange nachzudenken, um auf die einzelnen Werte zuzugreifen. Übrigens gilt dies eigentlich für alle Daten die in eZ gespeichert werden wie z.B. eZContentObject oder eZUser Objekte.

Listing 14. jacextension/classes/jacextensiondata.php Beispiel für den Datenbankzugriff über eZ PersistentObjekt

```
<?php
include_once( 'kernel/classes/ezpersistentobject.php' );
include_once( 'kernel/classes/ezcontentobject.php' );
include_once( 'kernel/classes/datatypes/ezuser/ezuser.php' );

class JACExtensionData extends eZPersistentObject
{
    /*!
       Konstruktor
    */
    function JACExtensionData( $row )
    {
        $this->eZPersistentObject( $row );
    }

    /*!
       Definition des Aufbaus des Datenobjektes / der
Datenbanktabellenstruktur
     */
    function definition()
    {
        return array( 'fields' => array(
       'id' => array(   'name' => 'ID',
              'datatype' => 'integer',
              'default' => 0,
```

```
                'required' => true ),
        'user_id' => array(    'name' => 'UserID',
                'datatype' => 'integer',
                'default' => 0,
                'required' => true ),
        'created' => array(    'name'  => 'Created',
                'datatype' => 'integer',
                'default' => 0,
                'required' => true ),
        'value' => array(       'name' => 'Value',
                'datatype' => 'string',
                'default' => '',
                'required' => true )
        ),
     'keys'=> array( 'id' ),
     'function_attributes' => array( 'user_object' => 'getUserObject' ),
     'increment_key' => 'id',
     'class_name' => 'JACExtensionData',
     'name' => 'jacextension_data' );
}

    /*!
       Hier erweitern wir attribute() aus eZPersistentObject
     */
    function &attribute( $attr )
    {
        if ( $attr == 'user_object' )
            return $this->getUserObject();
        else
            return eZPersistentObject::attribute( $attr );
    }

    /*!
       Hilfsfunktion wird in Funktion attribute aufgerufen
     */
    function &getUserObject( $asObject = true )
```

```
    {
        $userID = $this->attribute('user_id');
        $user = eZUser::fetch($userID, $asObject);
        return $user;
    }


    /*!
       erzeugt ein neue Objekt vom Typ JACExtensionData und gibt dieses
zurück
     */
    function create( $user_id, $value )
    {

        $row = array( 'id' => null,
                      'user_id' => $user_id,
                      'value' => $value,
                      'created' => time() );

        return new JACExtensionData( $row );
    }

    /*!
       liefert den Datensatz mit gegebener id als JACExtensionData Objekt
zurück
     */
    function &fetchByID( $id , $asObject = true)
    {
        $result = eZPersistentObject::fetchObject(

                          JACExtensionData::definition(),
                null,
                array( 'id' => $id ),
                $asObject,
                null,
                null );
```

```
     if ( is_object($result) )
         return $result;
        else
            return false;
    }

    /*!
       liefert alle JACExtensionData Objekte entweder als Objekt oder Array
zurück
     */
    function &fetchList( $asObject = true )
    {
  $result = eZPersistentObject::fetchObjectList(
       JACExtensionData::definition(),
       null,null,null,null,
       $asObject,
       false,null );
  return $result;
    }
    /*!
     liefert die Anzahl der Datensätze zurük
     */
    function getListCount()
    {
        $db =& eZDB::instance();
        $query = 'SELECT COUNT(id) AS count FROM jacextension_data';
        $rows = $db->arrayQuery( $query );
        return $rows[0]['count']
    }
    // -- member variables--
    //// \privatesection
    var $ID;
    var $UserID;
    var $Created;
    var $Value;
```

```
}
?>
```

Listing 15. jacextension/modules/modul1/create.php Erzeugen eines neuen Datenbankeintrags und verschiedene Möglichkeiten diese wieder auszulesen

```
<?php
// modul1/create.php - Funktionsdatei der View create
...
$value = '';
// Wenn Variable 'name' per GET oder POST übertragen wurde, Variable auslesen
if( $http->hasVariable('name') )
    $value =  $http->variable('name');
if( $value != '' )
{
    include_once('kernel/classes/datatypes/ezuser/ezuser.php');
    // die ID des aktuellen Users abfragen
    $userId = eZUser::currentUserID();

    include_once('extension/jacextension/classes/jacextensiondata.php');
    // neue Datenobjekt generieren
    $JacDataObject = JACExtensionData::create( $userId, $value );

            eZDebug::writeDebug(  '1.'.print_r(  $JacDataObject,  true  ), 'JacDataObject vor dem Speichern: ID nicht gesetzt') ;

    // Objekt in Datenbank speichern
    $JacDataObject->store();

            eZDebug::writeDebug(  '2.'.print_r(  $JacDataObject,  true  ), 'JacDataObject nach dem Speichern: ID ist gesetzt') ;

    // id des neu angelegten Objekte erfragen
    $id = $JacDataObject->attribute('id');

    // loginnamen des users erfragen, der den Datensatz angelegt hat
```

```
    $userObject = $JacDataObject->attribute('user_object');
    $userName = $userObject->attribute('login');

    // Datensatz neu auslesen
    $dataObject = JACExtensionData::fetchByID($id);

     eZDebug::writeDebug( '3.'.print_r( $dataObject, true ), 'JacDataObject
ausgelesen Ã¼ber Funktion fetchByID()');

    // Anzahl vorhandener DatensÃ¤tze ermitteln
    $count = JACExtensionData::getListCount();

    $statusMessage = 'Name: >>'. $value .'<< von Benutzer >>'.$userName.'<<
in Datenbank mit der id >>'.
                    $id.'<< gespeichert! Neue Anzahl = '.$count ;
}
else
{
    $statusMessage = 'Bitte Daten eingeben';
}

// Listendaten einmal als Objekte und einmal als Array holen und in den
Debug Output schreiben
$ObjectArray = JACExtensionData::fetchList(true);
eZDebug::writeDebug( '4. JacDataObjects: '.print_r( $ObjectArray, true ),
'fetchList( $asObject = true )');

$array = JACExtensionData::fetchList(false);
eZDebug::writeDebug( '5. JacDataArrays: '.print_r( $array, true ),
'fetchList( $asObject = false )');

// Templateobject initialisieren
$tpl =& templateInit();
$tpl->setVariable( 'status_message', $statusMessage );
...
```

```
?>
```

1.3.10 Template Fetch Funktion

Wir wissen nun wie Parameter, GET / POST Variablen an eine View übergeben und ausgelesen werden. Möchten wir aber in in irgendeinem Template von eZ publish z.B. Daten aus unserer Datanbanktabelle anzeigen, kommen wir mit dem direkten Aufruf der View nicht weiter. Für diesen Zweck gibt es die Fetch Funktionen wie wir sie von dem eZ Kernel modul kennen z.B. {fetch('content', 'node', hash('node_id', 2)}. Wir wollen uns zwei Fetch Funktion *list* und *count* definieren, die über folgende Templatesyntax aufgerufen werden können. {fetch('modul1', 'list', hash('as_object' , true()))} und {fetch('modul1', 'count', hash())}. Die *list* Funktion gibt alle Dateneinträge wahlweise als Array oder Objekte zurück, wird über den selbst definierten Parameter 'as_object' gesteuert. Die *count* Funktion hat keinen Parameter und ermittelt mit einer einfache SQL Anweisung die Anzahl der Datensätze unserer Beispieldatenbanktabelle *jacextension_data*.

Die Definition der Fetch Funktionen erfolgt in der *jacextension/modules/modul1/function_definition.php*. Hier wird festgelegt, welche Parameter an welche PHP Funktion einer bestimmten PHP Klasse übergeben werden (siehe Listing 16)

In der Datei *jacextension/modules/modul1/ezmodul1functioncollection.php* befindet sich eine Hilfsklasse, die alle Fetch Funktionen beinhaltet (siehe Listing 17) .

Tipp: In der *...functioncollection.php* eines jeden Moduls kann man die möglichen Parameter für den hash() Teil einer Fetch Funktion nachlesen. So kann z.B. jeder selbst in der Datei *kernel/content/ezcontentfunctioncollection.php* nachlesen, welche Parameter es für die Fetch Funktion {fetch('content', 'tree', hash(...))} des eZ Kernel Moduls *content* gibt. Das ist oft hilfreich, wenn die Online Dokumentation von eZ publish unvollständig ist.

Listing 16. Definition der Template Fetch Funktionen des Moduls modul1 - extension/jacextension/modules/modul1/function_defintion.php

```
<?php
$FunctionList = array();

// {fetch('modul1','list', hash('as_object', true()))|attribute(show)}
$FunctionList['list'] = array(
       'name' => 'list',
      'operation_types' => array( 'read' ),
      'call_method' =>
                array('include_file' =>
 'extension/jacextension/modules/modul1/ezmodul1functioncollection.php',
                     'class' => 'eZModul1FunctionCollection',
                     'method' => 'fetchJacExtensionDataList' ),
      'parameter_type' => 'standard',
      'parameters' => array( array( 'name' => 'as_object',
                                    'type' => 'integer',
                                    'required' => true ) )
   );
//{fetch('modul1','count', hash())}
$FunctionList['count'] = array(
     'name' => 'count',
    'operation_types' => array( 'read' ),
    'call_method' => array(
        'include_file' =>
 'extension/jacextension/modules/modul1/ezmodul1functioncollection.php',
        'class' => 'eZModul1FunctionCollection',
        'method' => 'fetchJacExtensionDataListCount' ),
    'parameter_type' => 'standard',
    'parameters' => array()
    );
?>
```

Listing 17. Hilfsklasse mit PHP Funktionen die in der Definition der Template Fetch Funktion in function_defintion.php genutzt werden – extension/jacextension/modules/modul1/ezmodul1functioncollection.php

```
<?php
include_once('extension/jacextension/classes/jacextensiondata.php');

class eZModul1FunctionCollection {
    function eZModul1FunctionCollection()
    {}
    // wird von fetch('modul1', 'list', hash('as_object', $bool ))
    // aufgerufen
    function fetchJacExtensionDataList( $asObject )
    {
        return array( 'result' => JACExtensionData::fetchList($asObject) );
    }
   // wird von fetch('modul1', 'count', hash() ) aufgerufen
   function fetchJacExtensionDataListCount( )
   {
       return array( 'result' => JACExtensionData::getListCount() );
   }
}
?>
```

1.3.11 Template Operator

Eine andere Möglichkeit auf Funktionen in einer eigenen Extension zuzugreifen, sind Template Operatoren. eZ publish bringt schon eine Vielzahl von Template Operatoren mit. Im folgenden definieren wir uns einen neuen Operator mit Namen *jac($result_type)* mit einem Parameter *result_type* . Über den Parameter wollen wir steuern, ob alle Datensätze unserer Tabelle oder nur die Anzahl zurückgeben werden soll. Der Tempalteaufruf {jac('list')} gibt ein Array von Datensätzen zurück und {jac('count')} die Anzahl der Datensätze. Dabei werden in unserem Beispiel die gleichen Funktionen, wie bei den Template Fetch

Funktionen fetch('modul1', 'list' , ...) und fetch('modul1', 'count', ...) verwendet. Die Definition der verfügbaren Templateoperatoren in unserer Extension jacextension erfolgt in der *extension/jacextension/autoloads/ eztemplateautoload.php* (siehe Listing 18). Was der Templateoperator macht, wird in einer eigenen PHP Klasse definiert. In unserem Fall in JACOperator in der Datei *extension/jacextension/autoloads/ jacoperator.php* (siehe Listing 19). Damit eZ publish weiß, dass unsere Extension *jacextension* Template Operatoren enthält, müssen wir dies in der Konfigurationsdatei *extension/jacextension/settings/site.ini.append.php* eZ publish über ExtensionAutoloadPath[]=jacextension mitteilen (siehe Listing 20).

Zum Testen unserer Template Fetch Funktionen fetch('modul1', 'list', hash('as_object', true())) und fetch('modul1', 'count', hash()), des Template Operators jac('list') oder jac('count') erweitern wir das Template *list.tpl* der View *list* (siehe Listing 21). Bei Aufruf der View z.B. über die URL *http://localhost/ez/index.php/plain_site/modul1/list/tableblue/1234* sollten nun neben der Ausgabe des Beispiel Array $data_array auch zwei mal die Datensätze der Datenbanktabelle *jacextension_data* angezeigt werden, einmal über die Template Operatoren und und ein zweites Mal über die Template Fetch Funktionen. Es gibt also mehrere Möglichkeiten auf die selben Funktionen der eigenen Extension in einem Template zuzugreifen.

Listing 18. extension/jacextension/autoloads/eztemplateautoload.php

```
<?php
// Welche Operatoren sollen automatisch geladen werden?
$eZTemplateOperatorArray = array();
// Operator: jacdata
$eZTemplateOperatorArray[]        =        array(        'script'        =>
'extension/jacextension/autoloads/jacoperator.php',
                                    'class' => 'JACOperator',
                                    'operator_names' => array( 'jac' ) );
?>
```

Listing 19. extension/jacextension/autoloads/jacoperator.php

```
<?php
/*!
 Operator: jac('list') und jac('count') <br>
 Count: {jac('count')} <br>
 Liste: {jac('list')|attribute(show)}
 */
class JACOperator{
    var $Operators;

    function JACOperator( $name = "jac" ){
     $this->Operators = array( $name );
    }

    /*!
     Returns the template operators.
    */
    function &operatorList(){
     return $this->Operators;
    }

    /*!
       \return true to tell the template engine that the parameter list
exists per operator type.
     */
    function namedParameterPerOperator()
    {
        return true;
    }

    /*!
     See eZTemplateOperator::namedParameterList
     */
    function namedParameterList()
    {
        return array( 'jac' => array( 'result_type' =>
                                      array(  'type' => 'string',
```

```
                                                    'required' => true,
                                                    'default' => 'list' )) );
    }

    /*!
        Je nachdem welche Parameter Übergeben worden sind JACExtensionData
Objekte fetchen {jac('list)}  oder Datansätze zählen  {jac('count')}
    */
    function modify( &$tpl, &$operatorName, &$operatorParameters,
                     &$rootNamespace, &$currentNamespace, &$operatorValue,
                     &$namedParameters){
       include_once('extension/jacextension/classes/jacextensiondata.php');

       $result_type = $namedParameters['result_type'];

       if( $result_type == 'list')
           $operatorValue = JACExtensionData::fetchList(true);
       elseif( $result_type == 'count')
           $operatorValue = JACExtensionData::getListCount();
    }
};
?>
```

Listing 20. extension/jacextension/settings/site.ini.append.php

```
<?php /* #?ini charset="utf-8"?
...
# nach Templateoperatoren in jaceextension suchen
[TemplateSettings]
ExtensionAutoloadPath[]=jacextension
*/ ?>
```

Listing 21. Testen der selbst definierten Template Fetch Funktionen und des Template Operators - extension/jacextension/design/standard/templates/modul1/list.tpl

```
<h2>Template Operator: jac('count') und jac('list')</h2>
Count: {jac( 'count' )} <br />
```

```
Liste: {jac( 'list' )|attribute(show)}
<hr>
<h2>Templatefetchfunktionen:
fetch('modul1','count', hash() <br />und<br />
fetch('modul1','list', hash( 'as_object', true()) )</h2>
Count: {fetch( 'modul1', 'count', hash() )} <br>
Liste:   {fetch(   'modul1',   'list',   hash(   'as_object',   true()))|
attribute(show)}
```

1.3.12 INI Datei

Abschließend wollen wir noch eine eigene Default .ini Datei anlegen *extension/jacextension/settings/jacextension.ini*. In diese lagern wir alle Werte aus, die wir in Templates oder Modulen fest eingetragen haben und die eventuell auf verschiedenen eZ Installation variieren können z.B. ob spezielle Debugmeldungen ausgegeben werden sollen. Die default *.ini* kann dann über die *jacextension.ini.append.php* Dateien z.B. im Siteaccess überschrieben werden. Listing 22 ist ein Beispiel für eine .ini Datei und Listing 23 zeigt wie in PHP auf diese zugegriffen werden kann. Wir erweitern *list.php*, um den Zugriff zu testen. Welche Funktionen es noch für den Zugriff auf INI Dateien gibt, kann in der API Dokumentation zu Klasse eZINI nachgelesen werden: *http://pubsvn.ez.no/doxygen/classeZINI.html*.

Abbildung 1 gibt noch einmal den Überblick über den Aufbau unseres Bespiels: *jacextension*.

Listing 22. Konfigurationsdatei der Extensin jacextension – extension/jacextension/settings/jacextension.ini

```
[JACExtensionSettings]
# Should Debug enabled / disabled
JacDebug=enabled
```

Listing 23. PHP Zugriff auf INI Datei jacextension.ini – extension/jacextension/modules/modul1/list.php

```
...
// Variable JacDebug aus dem INI Block [JACExtensionSettings]
// der INI Datei jacextension.ini lesen
include_once( "lib/ezutils/classes/ezini.php" );
$jacextensionINI =& eZINI::instance( 'jacextension.ini' );
$jacDebug = $jacextensionINI->variable('JACExtensionSettings','JacDebug');
// Wenn Debug eingeschaltet mache irgendwas
if( $jacDebug === 'enabled' )
    echo 'jacextension.ini: [JACExtensionSetting] JacDebug=enabled'; ...
```

1.3.13 Aufbau Beispielextension 'jacextension' & Sourcecode

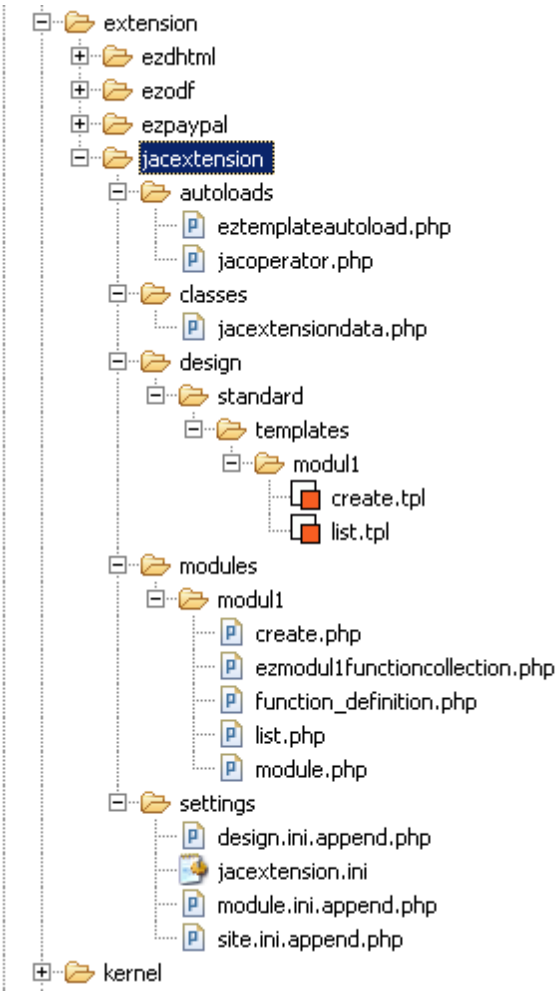

Abbildung 1: Verzeichnis-/Dateistruktur der Beispielextension jacextension

Der Sourcecode zum Tutorial kann im Internet unter folgender Adresse heruntergeladen werden:

http://ez.no/community/contribs/examples/jacextension

1.4 Fazit

Wir haben an einem einfachen Beispiel von *jacextension* die verschiedenen Techniken gelernt, die bei der Entwicklung von Extensions hilfreich sind. Neben dem Anlegen von eigenen Modulen mit verschiedenen Views und View Parametern, Template Fetch Funktionen, eines Template Operators und der Nutzung des Rechtesystems von eZ publish, wissen wir nun, wie eigene Meldungen in die Debug Ausgabe oder in eigene Log Dateien geschrieben werden können. Wie der Zugriff auf INI Dateien erfolgt, wurde abschließend kurz erläutert.

Mit diesem Grundlagenwissen sollte es jetzt einfach möglich sein, eigene Extensions zu entwickeln.

1.4.1 Links im Internet

- *http://www.ezpublish.de/* – Deutsche eZ Community
- *http://www.ez.no/community - Internationale eZ Community*
- *http://pubsvn.ez.no/doxygen/index.html* – eZ API Dokumentation
- *http://ez.no/doc/ez_publish/technical_manual/3_8/reference* – eZ Referenz Dokumentation
- http://ez.no/community/contribs/documentation/jac_dokumentation_in_german_ez_publish_basics_extension_development – PDF eZ publish Einführung / Grundlagen Modulprogrammierung

1.4.2 Über den Autor

Felix Woldt studierte Informatik an der FH-Stralsund. Während des Studium (seit 2002) und in seiner Diplomarbeit (2004) beschäftigte er sich mit eZ publish und der Programmierung von Extensions. Felix ist aktives Mitglied der deutschsprachigen eZ publish Community http://ezpublish.de und Mitarbeiter beim eZ Partner JAC Systeme http://www.jac-systeme.de.