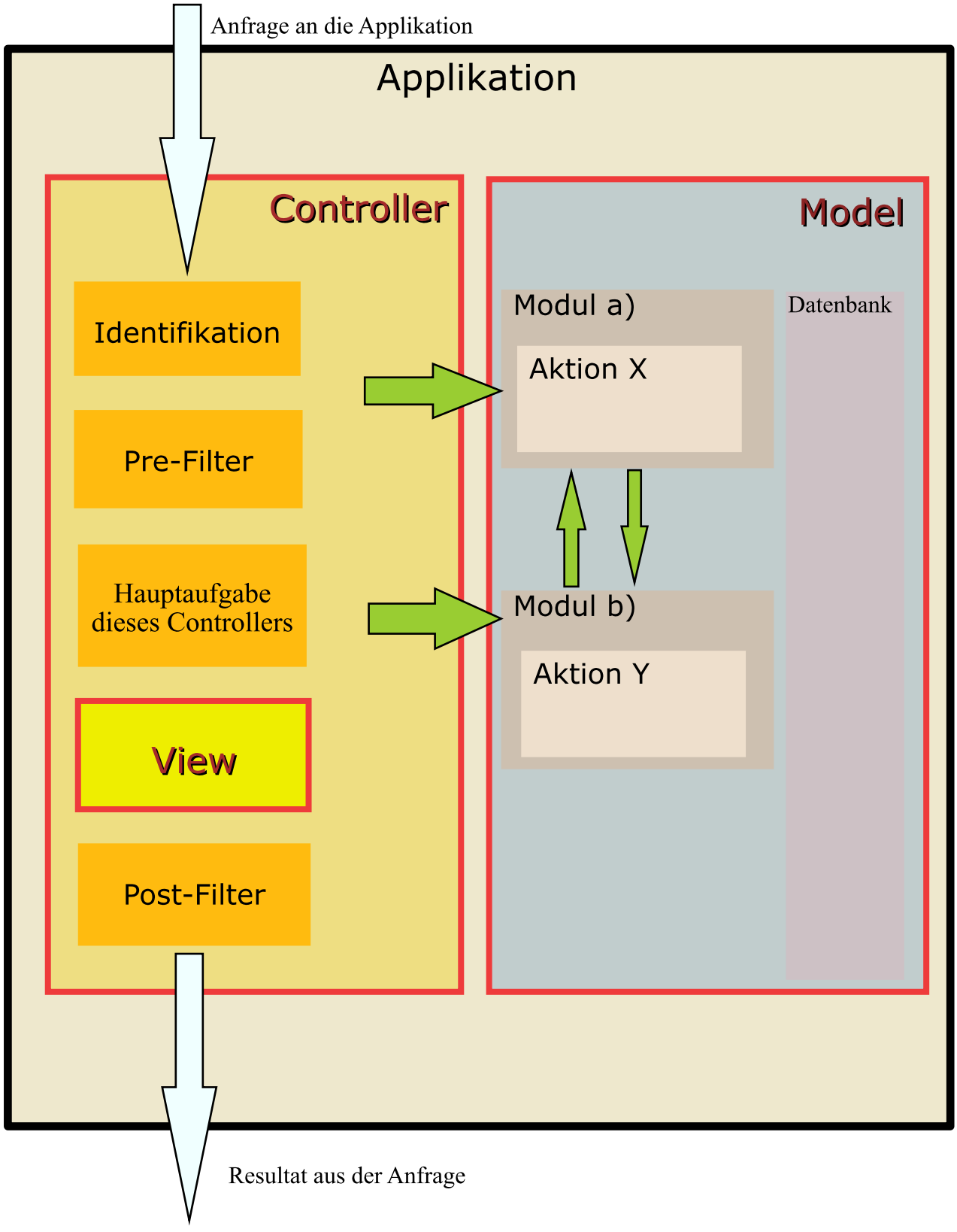
Dieses Dokument beschreibt das Applikationsdesign
des CMS Open Publisher.
http://www.open-publisher.net

Applikations-Design - version 1.0

Das Design folgt dem MVC Schema. Danach besteht die Architektur dieses Designs aus 3 Einheiten; Datenmodel (Model), Präsentation (View) und Programmsteuerung (Control).



Grundlage

Nun sind in MVC die 3 Element genau definiert aber nicht wie dieses in die Praxis umzusetzen ist. Deshalb weicht die oben gezeichnete Umsetzung von anderen Umsetzungen ab. Abweichungen gibt es was die Aufgabe des Controllers anbetrifft.

Die einzelnen Elemente werden nun anhand eines praktischen Beispiels erklärt. Bei der Applikation handelt es sich um eine Webseite an die die Anfrage gestellt wird: "Zeige mir einen Text mit einer bestimmten ID". Das System nimmt die Anfage entgegen und entscheidet welcher Controller für diese Anfrage zuständig ist. Denn je nach Anfrage ist die Aufgabe die ein Controller zu erfüllen hat anders. Z.B. Lautet die Anfrage "Zeige mir die Eingangsseite" so ist die Aufgabe die eine Controller zu erledigen hat eine völlig Andere. Für das obrige Beispiel nennen wir den Controller "ShowArticle"

Der (Das) die Anfrage stellt wird zuerst identifziert. Also hat ein Login stattgefunden oder nicht (Session).

Durch die Pre-Filter können für diese Anfrage eine Reihe von speziellen Opperationen durchgeführt die je nach Controller anders aussehen können. Z.B. Spam Robots Filter, Logging. Nun kann man darüber streiten was ein Filter machen soll oder nicht. Filter werden vor der eigentlichen Hauptaufgabe des Controllers aktiv.

Jetzt folgt die eigentliche Aufgabe des Controllers die daraus besteht den Text mit der bestimmten ID aus der Datenbank zu beschaffen und Templatevariablen (View) mit seinem Inhalt zu füllen. Dafür greift der Controller nicht direkt auf die Datenbank zu sondern über den Umweg des Model. Der Controller führt eine Action des Models aus die darauf spezialisiert ist Text aus der Datenbank auszulesen. Dabei stellt jedes Modul eine Teil des Models. Z.B. stellt das Modul "Text" Aktionen zur Verfügung die zum auslesen bzw verändern von Textdaten dienen. Ein Anderes Modul würde Aktionen enthalten was Userdaten anbelangt. Ein anderes Modul was die Navigationsknoten anbetrifft.

Anschliessend lädt der Controller die zu ihm zugehörige View (Template). Die Viewvariablen werden dort eingefügt.

Post-Filter runden das Ganze ab. Z.B. aus der View Html-Kommentare ausfiltern.

In jeder Etape des Controllers können beliebig viele Modelaktionen ausgeführt werden. In der View sollte (darf) das nicht möglich sein. Also in der view geht es nur um die Darstellung der Daten und nicht um Programlogik. Dafür sollte es möglich sein, dass eine Aktion eine andere Aktion aufrufen kann.

Desweiteren sollte es möglich sein, dass Controller hierarchisch verkettet werden können. D.h. dass z.B. der Controller "ShowArticle" den Controller einbindet der sich um die Darstellung der Haupt-Navigationsknoten der Webseite kümmern soll. Sowie den Footer, Header, Banner und was immer sonst noch ansteht.

Weiterhin sollten die einzelnen Elemente eines Controllers optional sein. Denn es ist nicht immer sinnvoll Filterfunktionen oder Views auszuführen.

Ordnerstruktur

- **application**
  - cache
  - configs
  - controllers **(controller)**
    - default
  - view_helper
  - logs
- **library**
  - japa (core framework)
- **modules**
  - common
    - actions **(model)**
    - controllers
    - views
  - default
    - actions **(model)**
    - controllers
    - views
- **public**
  - images
  - scripts
  - styles
  - views **(view)**
    - default

.htaccess

index.php (bootstrap)

Im Ordner **application/controllers** liegen u.a. alle Controller die für das Erstellen eines Projektes notwendig sind. (Ohne die Controller des Admininterfaces)

Der Ordner **application/configs** enhält Konfigurationsdateien.

application/logs enthält eventuelle Meldungen die das System ausgibt. error_log, usw

application/view_helper enthält Hilfs-Funktionen die in Views eingesetzt werden können

library enthält das core framework und alle weiteren libraries pear, zend oder was auch immer.

modules enhält alle Module. Dabei bringt jedes Modul folgende Komponenten mit: Actions die Teil des **Models** sind. Controller und Views für das Administrations-interface. Diese beiden Komponenten sind der Übersicht halber nicht in der Grafik enthalten.

2 Module sollten mindestens enthalten sein.

1) **common** - Bei allen Anfragen die an die Administration gehen wird zuerst der Indexcontroller dieses Moduls geladen. Dieser lädt dann den Indexcontroller des Moduls an die die Anfrage eigentlich gehen soll. Ist kein bestimmtes Modul in der Anfrage definiert so wird der Indexcontroller des 2) **default** Moduls geladen.

Über index.php (bootstrap) gehen alle Anfragen an das System.

.htaccess leitet alle http Anfragen auf index.php weiter.

Routing (url rewrite)

Alle http Anfragen werden durch die in der .htaccess Datei definierte Regeln auf die index.php weitergeleitet. Dazu muss das Apachemodul mod_rewrite aktiviert sein.

Ist im Url nichts definiert. Also z.B.

http://www.test.com

so lädt das System den Indexcontroller in:

/application/controllers/default/ControllerIndex.php

Soll ein anderer Controller geladen werden so kann dieses in der Url wie folgt definiert sein:
Beispiel:
http://www.test.com/cntr/example2

Also **cntr** steht für Controller wobei nachfolgend der Name des Controllers steht: **Example2**
Der entsprechende Controller ist:

/application/controllers/default/Expample2Controller.php

Der Router teilt den Url auf, so dass auf den Name einer Variable deren Inhalt folgt. Alle Variablen werden im Array $request gespeichert.
Beipiel:
http://www.test.com/cntr/article/id_article/4325

Das $request Array des Routers enthält dann die Variablen:

```
array("cntr"       => "article",
      "id_article" => "4325")
```

Sollen die Controller des Admininterfaces angesprochen werden so muss im Url der Abschnitt "Module" vorkommen:
http://www.test.com/Module
Es werden daraufhin die Controller und Views der Module geladen.
Es passiert nun folgendes:
Das System lädt zuerst den Indexcontroller des "common" moduls:

/modules/common/controllers/ControllerCommonIndex.php

Dieser Controller lädt dann den eigentlichen Modulcontroller. Ist solch einer nicht definiert so wird der Indexcontroller des Default Moduls geladen.

/modules/default/controllers/ControllerDefaultIndex.php

Soll der Controller eines anderen Moduls geladen werden so sieht die url z.B. so aus:
http://www.test.com/Module/mod/article/cntr/edit/id_article/5436

hier wird der Controller "edit" des Article Moduls geladen um den Artikel mit der ID 5436 zu bearbeiten. Also der controller:
/modules/article/controllers/ControllerArticleEdit.php

Der Routingmechanismus kann aber noch weit mehr. Jedes Modul kann entscheiden ob es für eine gegebene Anfrage automatisch den Namen des zuständigen Controller bereit hält. So wird wenn z.B. die Variable id_article in einer Anfrage vorhanden ist vom System das Artikelmodul darüber informiert und gegebenfalls gibt dieses Modul eine Antwort zurück welcher Controller für solch eine Anfrage zuständig ist.

Desweiteren lassen sich so lesbare URL's erzeugen. Also z.B. anstelle von:
http://www.test.com/id_article/6
der URL:
http://www.test.com/Einfuhrung

Es brauch hierzu keine Veränderungen in der .htaccess Datei vorgenommen zu werden.

Controller

Bei den Controllern die zum Erstellen eines Projektes benutzt werden handelt es sich um Seitencontroller (PageController). Die php-Klasse dieser Controller enthält einige Methoden die vom Applikationscontroller des Systems in einer bestimmten Reihenfolge aufgerufen werden. Alle Methoden sind optional.

auth()
Hier kann die Authentifizierung stattfinden.

prependFilterChain()
Hier können Filter eingebaut werden die bevor der eigentlichen Hauptfunktion des Controllers wirksam werden.

perform()
Die Hauptmethode des Controllers

appendFilterCain()
Hier können filter eingebaut werden die nach der Haupfunktion des Controllers wirksam werden können. Z.B. Filter die auf der gerenderten View (Template) arbeiten.

Es stehen in dieser Klasse folgende Variablen zur Verfügung über die man einfluss auf das Verhalten des Systems hat:

public $viewVar
Dieses ist ein Array in dem alle Variablen abgespeichert werden die in der View eingefügt werden sollen.

public $controllerVar
Dieses ist ein Array in dem Variablen abgespeichrt werden können die in nested Controllern benutzt werden. Controller können ineinander verschachtelt werden.

public $model
Dieses ist eine Kopie der Model Instanz. Diese enthält später u.a. eine Datenbank und Sessioninstanz. Also Alles was das Model betrifft. Es gibt in der Modelklasse 2 Methoden die besonders wichtig sind.
action() und **broadcast()** über die die Kommunikation mit dem Model stattfindet. dazu später mehr im Kapitel Model.

public $config
Dieses ist das globale Konfigurationsarray.

public $renderView
Variable vom Type bool. Soll eine View gerendert werden oder nicht?

public $view
Per default stimmt der Name der View mit dem des Controller übereinander. Also heisst der Controller z.B. ControllerTest.php so wird die dazugehörige View view.Test.php gerendert. In dieser Variable kann der Name geändert werden.

public $returnView
Bei verschachtelten Views ist es sinnvoll, dass die gerenderte View nicht sofort vom System in den Ausgangsbuffer geschrieben sondern einfach nur intern zwischengespeichert wird.

public $viewEngine
Den namen der Viewengine (Templateengine) die für das Renderen der View zuständig ist. Die Defaultengine ist php.

$cacheExpire
Zeit in Sekunden danach der Cache gelöscht wird.

$cacheId
Die ID des cache. wird keine ID angegeben so erstellt das System eine ID.

$controllerLoader
Durch diese Instanz ist es möglich andere Controller auszuführen.
Z.B durch
$controllerLoader->test()
wird der Controller Controller.Test.php ausgeführt.

Aufbau eines Controllers:

```
class ControllerT extends JapaPageController
{
    public $viewVar = false;
    public $controllerVar = false;
    public $model;
    public $config;
    public $viewEngine = false;
    public $view = '';
    public $renderView = true;
    public $returnView = false;
    public $cacheExpire = 0;
    public $cacheId = false;
    public $controllerLoader;

    public function auth()
    {   }

    public function perform()
    {   }

    public function prependFilterChain()
    {   }

    public function appendFilterChain( & $viewBufferContent )
    {   }
}
```

View

Die View ist eine Datei in der Daten die vom Controller aufbereitet wurden dargestellt werden sollen. In einer klassischen Webseite ist das eine html Seite. Die View dient also als Vorlage (Template).

Daten die vom Controller in dem Array $this->viewVar abgespeichert wurden stehen in der View im Array $view zur Verfügung insofern PHP als Templateengine zum Einsatz kommt.

Aus Sicherheitsgründen ist es sinnvoll nicht jedes php Konstrukt in Views zuzulassen. Das wäre z.B. der Fall wenn Views nicht selbst erstellt werden. Im globalen Konfigurationsarray kann man durch aktivieren des Codeanalyzers das System dazu bewegen die Views auf darin enthaltene php-Konstrukte zu untersuchen.

$JapaConfig['useCodeAnalyzer'] = true;

In den nachfolgenden 2 Arrays können die Konstrukte sowie die Variablen festgelegt werden.

$JapaConfig['allowedConstructs']
$JapaConfig['disallowedVariables']

Die View wird vom System nach der Controllermethode perform() und bevor der Methode prependFilterChain() gerendert.

Letztere Methode kann dazu dienen um die gerenderte View nachzubearbeiten.

```
<html>
<head>
 <title><?php echo $view['title']; ?></title>
</head>

<body>
<h1><?php echo $view['title']; ?></h1>

<div><?php echo $view['text']; ?></div>

</body>
</html>
```

Model

Das Model besteht aus Actionklassen die in den jeweiligen /actions Ordner der Module liegen. Diese Klassen können über 2 Methoden (Model-Interface) der Modelklasse aufgerufen werden.

Aufruf einer einzigen Klasse:

```
$model->action('modul name',
                    'klassen name',
                    $daten);
```

Der erste Parameter enthält den Name des Moduls zum dem die Actionklasse zugehört. Der zweite Parameter enthält den Name der Klasse. der dritte Parameter kann Daten enthalten die zum korrekten ausführen der Klasse notwendig sind.

Beispiel der Klasse:

/modules/common/actions/ActionCommonFilterHtmlOutput.php

Mit dieser Klasse kann man aus Views (Templates) verschiedene Dinge ausfiltern:

```
$model->action('common', 'FilterHtmlOutput',
                array('str'      => & $template,
                      'filters' => array('stripComments',
                                         'trimOutput') ));
```

Das Model erstellt von dieser Klasse eine Instanz. Der dritte Parameter enthält ein Array das wiederum die Variable 'str' enhält. Hier wird dieser Variable eine Referenz auf die Variable übergeben die die komplette gerenderte View als String enthält. In 'filters' werden die Filter definiert die auf den String angewant werden. Es werden hier Kommentare ausgefiltert und Leerzeichen bevor und hinter der View ausgefiltert.

Solch ein Filter könnte z.B in den Controllermethoden prependFilterChain() und appendFilterChain() angewand werden. Aber nicht nur; Actionklassen können mit Ausnahme der View überall im System aufgerufen werden. Also auch von innerhalb von den Actionklassen selbst. Es können somit Klassen erstellt werden die wiederum von sich aus andere Actionklassen aufrufen.

Die Frage stellt sich jetzt von selbst; Warum man nicht selbst eine Instanz von solchen Klassen erstellen und die darin enthaltenen Methoden aufrufen kann?

Die Antwort ist, weil sich somit eine Plug&Play Struktur erzeugen lässt. Wird eine Actionklasse aufgerufen die nicht existiert so läuft das System weiter ohne Fehlermeldung.

Damit eine Klasse registriert wird braucht sie nur in den entsptrechenden /actions Ordner eines Module kopiert zu werden. Man braucht sich also weder um das include noch um die Erzeugung einer Instanz zu kümmern.

Wird eine Actionklasse aufgerufen so führt das System zuerst die validate() und dann die perform() Methode dieser Klasse aus.

Der Sinn der validate()

Methode besteht darin Daten die an die Klasse übergeben werden auf ihre Gültigkeit hin zu überprüfen. Diese Methode muss true zurückgeben damit die perform() Methode ausgeführt wird.

Es gibt noch eine zweite Möglichkeit das Model anzusprechen. Ein verteilter Aufruf (**broadcast**):

```
$model->broadcast('klassen name',
                  $daten);
```

Hier wird an alle Module ein Actionaufruf erteilt insofern ein Modul eine entsprechende Klasse bereithält. Das System selbst macht davon gebrauch:

```
$model->broadcast( 'init' );
```

Stellt ein Modul eine Actionklasse 'init' zur Verfügung so wird diese ausgeführt.

wird fortgesetzt ...