

Ein Template-System im Eigenbau

Im Zeitalter von Smarty & Co. mutet ein eigenes Template-System zu entwickeln fast wie das Rad neu zu erfinden an. Dennoch gibt es manchmal Fälle und gute Gründe auf solide Hausmannskost aus eigener Küche zu setzen. Welche diese sind und wie ein Ansatz aussehen kann, klärt dieser Artikel.

In fast jedem PHP Lehrbuch oder PHP Tutorial findet man den Rat, Darstellung und Programm-Code grundsätzlich zu trennen. Dies ist ein Ratschlag, den jeder erfahrene PHP Programmierer beherzigt. Die Wege, die dabei aber gegangen werden, um dieses Ziel zu erreichen, könnten unterschiedlicher nicht sein. Angefangen bei einem simplen `str_replace()`, bis hin zu komplexen Template-Engines, welche den View-Teil innerhalb des Model-View-Controller Patterns implementieren, hat der Software-Entwickler die Qual der Wahl.

Warum sollte man also das Rad neu erfinden? Im Internet finden sich zuhauf frei verfügbare Template-Systeme, welche alle mehr oder weniger komplex und ausgereift sind. Weiterhin bietet sich mit Smarty [1] eine Template-Engine an, welche gut dokumentiert und vom PHP Entwicklerteam sogar empfohlen wird. Schauen wir uns ein paar Gründe an, welche dennoch gegen den Einsatz „externer“ Template-Systeme sprechen.

- Lizenzprobleme, manche Kunden bestehen explizit auf 100% Eigenentwicklungen
- Code entspricht nicht den vorgegebenen Coding-Richtlinien (Benennung von Variablen / Dateien, Methoden, etc...)
- Komplexität, viele Template-Systeme sind zwar umfangreich, diese „Featuritis“ wird selten gebraucht. In vielen Fällen geht dies auf Lasten der Geschwindigkeit.
- Insbesondere wenn die Template-Engine eine eigene Skriptsprache mitbringt, ist das Debuggen der Templates oftmals sehr aufwändig.
- Anpassungen am Fremdcode sind im Falle von Fehlern oder Erweiterungswünschen nicht immer einfach zu implementieren.

Was sollte ein Template-System leisten?

Eigentlich nicht viel. Es sollte eine Möglichkeit bieten, Code und Design sauber zu trennen und dabei nach Möglichkeit einfach benutzbar bleiben. Letzteres bezieht sich dabei sowohl auf den PHP Entwickler als auch auf den Designer, der unter Umständen die grafischen HTML-Arbeiten durchführt. Und wie bei jedem zusätzlichen Stück Code gilt auch letztendlich hier: je schneller das Ganze, desto besser.

Einfachheit für den HTML-Designer

Ein häufig angeführtes Argument ist, dass eine Template-Engine in der HTML-Datei eine, für den HTML-Designer, möglichst einfache Syntax bieten sollte, um Fallunterscheidungen oder Schleifen zu ermöglichen. Aus diesem Grund führen manche Template-Systeme eine eigene Syntax ein. Dass dies zu Lasten der Geschwindigkeit geht, schliesslich muss ein richtiger Parser den Code im Template ausführen, leuchtet ein. Aber auch hier wurden Lösungen gefunden. Manche Template-Systeme führen diesen Schritt einmalig durch und cachen das Ergebnis für nachfolgende Aufrufe. Etwas irreführend sprechen die Dokumentationen häufig dann von „kompilierten Templates“. Schaut man sich das Kompilat jedoch an, wird man feststellen, dass sich hier letztendlich eine PHP-Datei verbirgt, deren Ausführung natürlich schneller erfolgt. Doch ist diese Vorgehensweise wirklich notwendig? Besinnen wir uns einen Schritt zurück. Die erste Version von PHP wurde von Rasmus Lerdorf entwickelt, weil er das Verlangen nach Dynamik in HTML-Dateien verspürte. Also

entwickelte er eine Skript-Sprache, welche sich leicht in HTML-Dateien einbetten ließ. Fällt Ihnen etwas auf? Aus dem Wunsch heraus, möglichst einfach zu sein, implementieren manche Template-Systeme eine Skriptsprache in einer Skriptsprache.

Ist aber die verwendete Syntax wirklich viel einfacher? Dazu ein profanes Beispiel aus Smarty:

```
{if $login eq "Felix"}
  Hallo Felix – schön dich zu sehen!
{else}
  Zugriff verweigert.
{/if}
```

Benutzt man hierfür die alternative PHP Syntax [2] im HTML Template, sieht obiges Beispiel wie folgt aus:

```
<?if ($login=="Felix"):?>
  Hallo Felix – schön dich zu sehen!
<?else:?>
  Zugriff verweigert.
<?endif;?>
```

Hier scheiden sich die Geister, was denn nun einfacher zu benutzen ist. Meiner Meinung nach dürften für einen Web-Designer beide Versionen gleich problematisch oder unproblematisch sein. Falls die Template-Erstellung ebenfalls zum Aufgabenbereich des PHP-Entwicklers gehört, ist letzteres Beispiel sicherlich die bessere Wahl - ob mittels klassischer Syntax oder verkürzt -, schliesslich muss sich dieser nicht noch zusätzlich mit einer weiteren Skriptsprache befassen.

Erster Ansatz

Berücksichtigen wir unsere Prämisse - saubere Trennung von Darstellungslogik und Code - ergibt sich recht bald ein Problem. PHP bietet zur Zeit leider keine Patentrezepte an, Variablen nur in einem bestimmten Kontext zugänglich zu machen. Aus anderen Programmiersprachen bekannte Konzepte, wie zum Beispiel Namespaces, sind zukünftigen PHP Versionen vorbehalten. Diesen Umstand können wir jedoch umgehen, indem wir unser Templatesystem in ein Objekt kapseln (Siehe Listing 1). Der dabei zu Grunde liegende Gedanke ist, der Template-Klasse die Variablen, welche im Template verfügbar sein sollen, als Eigenschaften zuzuweisen. Da das Template per `include()` innerhalb der Methode `_parse()` geladen wird, sind nur die Klasseneigenschaften im Template verfügbar (und natürlich etwaige, innerhalb der Methode `_parse()` definierte, Variablen).

Listing 1. Eine rudimentäre Template-Klasse

```
<?php
class fggTemplate
{
  public function setVar($cName,$xVal)
  {
    $this->$cName = $xVal;
  }

  public function getContent($cTemplate)
  {
    return $this->_parse($cTemplate);
  }
}
```

```

public function display($cTemplate)
{
    echo $this->_parse($cTemplate);
}

protected function _parse($cTemplate)
{
    ob_start();
    include($cTemplate);
    return ob_get_clean();
}
}
?>

```

Die Benutzung im Code ist denkbar einfach. Nach dem Einbinden der Klassendatei wird ein Objekt vom Typ der Klasse instanziiert. Nun können mittels der Methode `setVar()` die benötigten Variablen im Template verfügbar gemacht werden, wobei der erste Parameter den Namen der Variable im Template und der zweite Parameter dessen Wert angibt. Mittels der Methode `display()` kann das Template ausgegeben werden, mittels der Methode `getContent()` kann der bereits geparste Code für eine weitere Verarbeitung gespeichert werden, wobei beide Methoden die anzuzeigende Template-Datei als Parameter erwarten. Listing 2 zeigt die Benutzung, Listing 3 das dazugehörige HTML-Template. In diesem wäre noch zu bemerken, dass der Zugriff via `$this->varname` erfolgt, da es sich um Eigenschaften der Klasse handelt.

Listing 2. Benutzung der Template-Klasse.

```

<?php
require_once("fggTemplate.inc.php");

$cTitle = "Eine kleine Buchübersicht";

$aBooks = array(
    array(
        "author" => "Bertolt Brecht",
        "title" => "Mutter Courage und ihre Kinder"
    ),
    array(
        "author" => "George Orwell",
        "title" => "1984"
    ),
    array(
        "author" => "Goethe",
        "title" => "Faust"
    )
);

$oTemplate = new fggTemplate();

$oTemplate->setVar("aBooks",$aBooks);
$oTemplate->setVar("cTitle",$cTitle);

$oTemplate->display("template.tpl.php");

```

?>

Listing 3. HTML Template

```
<html>
<head>
  <title><?=$this->cTitle?></title>
</head>

<body>
  <h2><?=$this->cTitle?></h2>

  <table border="1" width="600">
    <tr>
      <th>Autor</th>
      <th>Titel</th>
    </tr>
    <?php foreach ($this->aBooks as $aBook): ?>
      <tr>
        <td width="250"><?=$aBook["author"]?></td>
        <td width="350"><?=$aBook["title"]?></td>
      </tr>
    <?php endforeach; ?>
  </table>

</body>
</html>
```

Eine kleine Buchübersicht

Autor	Titel
Bertolt Brecht	Mutter Courage und ihre Kinder
George Orwell	1984
Goethe	Faust

Abbildung 1: Unser einfaches Template-System in Aktion

Damit hätten wir schon ein quick'n dirty Template-System, welches sogar zum jetzigen Zeitpunkt einsetzbar ist. Dadurch, dass wir innerhalb der Templates auf PHP gesetzt haben, haben wir uns sehr viel Programmierarbeit vom Hals gehalten und dabei Einiges an Geschwindigkeit gewonnen. Sub-

Templates übrigens, lassen sich auch ohne Probleme einbinden, hierfür ist im Haupt-Template ein einfaches `include()` zu benutzen.

Kleine Verbesserungen

Ein bisschen Komfort stünde unserer Template-Klasse sicherlich nicht schlecht zu Gesicht. Momentan müssen wir dem Template-System den Pfad zum Template direkt beim Aufruf von `display()` bzw. `getContent()` übergeben und auch das Benutzen von `$this->` im Template selbst mag der Eine oder Andere als störend empfinden. Zeit, dies zu ändern. Die fertige Klasse finden Sie in Listing 4, die Benutzung selbst in Listing 5.

Hierfür führen wir ein Array namens `_aConfig` als private Eigenschaft ein, welches die - in Zukunft zahlreichen - Konfigurationsoptionen aufnimmt. Das Setzen dieser Optionen erfolgt entweder direkt im Konstruktor des Objektes oder per entsprechenden „set-Methoden“. Momentan sind zwei solche Methoden implementiert, die Methode `setGlobals()` und die Methode `setTemplateDirs()`. Beiden gemeinsam ist, dass diese den Wert der entsprechenden Konfigurationsoption entgegennehmen und im weiter oben erwähnten Konfigurationsarray ablegen. Ein kleines Schmäckerl bietet noch die Methode `setTemplateDirs()`. Der Parameter kann entweder ein Array mit Verzeichnissen sein, oder ein String mit komma-separierten Werten.

Listing 4. Verbesserte Template-Klasse

```
<?php
class fggTemplate
{
    private $_aConfig;

    public function __construct(array $aConfig = null)
    {
        if (is_null($aConfig))
            $aConfig = array();

        $this->_aConfig = array("templateDirs"=>array(),
                               "globals"=>false);

        if (array_key_exists("templateDirs",$aConfig))
            $this->setTemplateDirs($aConfig["templateDirs"]);

        if (array_key_exists("globals",$aConfig))
            $this->setGlobals($aConfig["globals"]);
    }

    public function setTemplateDirs($xDir)
    {
        if (!is_array($xDir))
            $xDir = explode(",",$xDir);

        $this->_aConfig["templateDirs"] = array_merge(
            $this->_aConfig["templateDirs"],
            $xDir);
    }
}
```

```

public function setGlobals($bGlobals = true)
{
    if (is_bool($bGlobals))
        $this->_aConfig["globals"] = $bGlobals;
}

public function setVar($cName,$xVal)
{
    $this->$cName = $xVal;
}

public function getContent($cTemplate)
{
    $cTemplate = $this->_findTemplate(
        array_reverse($this->_aConfig["templateDirs"]),
        $cTemplate);
    return $this->_parse($cTemplate);
}

public function display($cTemplate)
{
    $cTemplate = $this->_findTemplate(
        array_reverse($this->_aConfig["templateDirs"]),
        $cTemplate);
    echo $this->_parse($cTemplate);
}

protected function _findTemplate(array $aDirs,$cFile)
{
    $cRet = null;
    $cBasePath = dirname(__FILE__);
    if (file_exists($cBasePath."/".$cFile))
        $cRet = $cBasePath."/".$cFile;
    else
        foreach ($aDirs as $cDir)
        {
            if (file_exists($cDir."/".$cFile))
            {
                $cRet = $cDir."/".$cFile;
                break;
            }
        }
}

if (is_null($cRet))
    die("Error: Did not find ".$cFile." :-(");

return $cRet;
}

protected function _parse($cTemplate)
{

```

```

    if ($this->_aConfig["globals"])
        extract(get_object_vars($this), EXTR_REFS);
    ob_start();
    include($cTemplate);
    return ob_get_clean();
}
} // EOC
?>

```

Listing 5. Benutzung der verbesserten Template-Klasse (Ausschnitt).

```

<?php
...
$oTemplate = new fggTemplate();
$oTemplate->setGlobals(true);

$aTemplateDirs = array ("/home/felix/templates1/",
                       "/home/felix/templates2/");
$oTemplate->setTemplateDirs($aTemplateDirs);

$oTemplate->setVar("aBooks",$aBooks);
$oTemplate->setVar("cTitle",$cTitle);

$oTemplate->display("template.tpl.php");
?>

```

Die Umsetzung der neuen Features ist ebenfalls so einfach wie möglich gehalten. Sofern eine Globalisierung im Template erwünscht ist, werden in der Methode `__parse()` mittels der PHP-Funktion `extract()` die Eigenschaften des Objektes als Variablen im aktuellen Gültigkeitsbereich zugänglich gemacht. Dies bezieht sich nur auf den Gültigkeitsbereich der Methode `__parse()` so dass unser superglobaler Gültigkeitsbereich nach wie vor unberührt bleibt. Somit können im Template nun die Variablen direkt mittels `$varname` angesprochen werden, falls der Entwickler sich an `$this->varname` stört.

Ebenfalls neu hinzugekommen ist die Methode `__findTemplate()`, welche nun von den Methoden `display()` und `getContent()` aufgerufen wird. Die Methode `__findTemplate()` prüft zunächst, ob sich das gewünschte Template evtl. direkt im Verzeichnis der Klassendatei befindet. Sollte dies nicht der Fall sein, geht sie die übergebenen Verzeichnisse schrittweise durch, bis ein passendes Template gefunden wurde. Die Prüfung der Verzeichnisse erfolgt übrigens in umgekehrter Reihenfolge des Setzens selbiger (Last in – first Out Prinzip). Beachten Sie bitte den Aufruf der Methode `__findTemplate()` in den Methoden `display()` und `getContent()`.

Fazit

Sicherlich ist unser Template-System noch längst nicht fertig, eine vernünftige Fehlerbehandlung mittels Exceptions wäre sicherlich wünschenswert. Auch weitere Features wie zum Beispiel ein flexibles Plugin-System könnten die spätere Arbeit mit dem System vereinfachen.

Ich bin mir sicher, dass Sie noch jede Menge Ideen für Erweiterungen haben. Ziel des Artikels war es jedoch, Ihnen die Entwicklung eines eigenen Template-Systems nahe zu legen und eine mögliche Vorgehensweise aufzuzeigen. Hierfür ist der umfangreich kommentierte Quellcode auf der Heft-CD ein guter Einstieg.

Im Internet

- <http://smarty.php.net/> - Die Smarty Web-Site [1];
- <http://www.php.net/manual/en/control-structures.alternative-syntax.php> – Alternative PHP Syntax [2]

Über den Autor

Felix-Gabriel Gangu beschäftigt sich seit 9 Jahren mit PHP und durfte in dieser Zeit an zahlreichen Projekten mitwirken. Er ist freiberuflicher Entwickler mit Fokus auf enterprise-ready Web-Applications sowie PHP Trainer und PHP Coach.

Kontakt mit dem Autor:

felix@gangu.de sowie <http://www.gangu.de>