Webrontend Development

Angular 2.0

Template-Syntax

Das Templating mit AngularJS war bereits ein mächtiges Werkzeug. Mit Angular 2.0 legen die Entwickler nun kräftig nach.

**Hinweis** Das hier gezeigte Beispiel nutzt eine Vorschauversion von Angular 2.0. Der gezeigte Code muss für spätere Versionen wahrscheinlich angepasst werden.

Einleitung

Im vorangegangenen Artikel "Angular 2.0 und modularer Code" wurde erläutert, wie mit `SystemJS` Bibliotheken und eigene Client-Side-Skripte geladen und ausgeführt werden können. Ein erstes Hello-World Beispiel mit Angular 2.0 wurde entwickelt. Das heißt, dass die Ausführung von ECMAScript 6 Modulen nun keine Hürde mehr ist. Es wird Zeit tiefer in das Framework einzutauchen.

Dieser Artikel stellt die neue Template-Syntax von Angular 2.0 vor. Es halten zahlreiche, neue Möglichkeiten Einzug, um Oberflächenelemente zu beschreiben. Die Entwickler von Angular verfolgen hierbei ein großes Ziel: Das Konzept der Template-Syntax eindeutiger und nachvollziehbarer zu formulieren, als es bei der Vorgängerversion der Fall ist. Auch der Support durch Editoren, etwa durch bessere automatische Vervollständigung, soll nun verbessert werden und die Produktivität des Entwicklers steigern.

Zur näheren Erläuterung wird ein Prototyp genutzt, der als Dashboard für Schäden an Autos dienen soll.

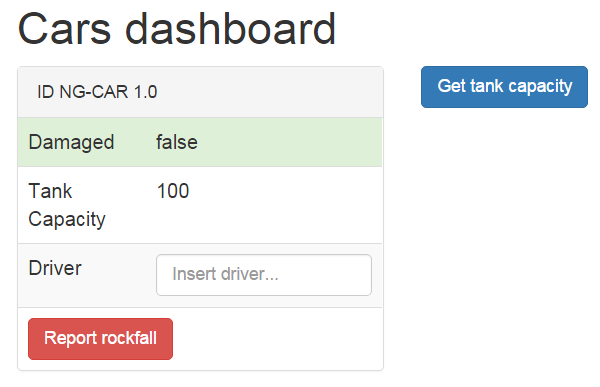


Abbildung 1 Dashboard Prototyp

Neben einer ID und dem Schadensstatus kann auch der aktuelle Füllstand des Fahrzeugs abgefragt werden. Des Weiteren kann mit einem Klick ein Steinschlag (engl: "rockfall") gemeldet werden.

**Übrigens** Sie finden das hier vorgestellten Beispiel auf GitHub unter <https://github.com/Angular2Buch/template-syntax>.

Components & Views

Angular 2 Anwendungen bestehen aus verschiedenen Komponenten (Components), die miteinander agieren können. Für das Dashboard wird eine Komponente benötigt. Im Dashboard wird eine Liste von Autos abgebildet werden. Das bedeutet, dass hierfür eine weitere Komponente implementiert wird.

Eine Angular 2.0 Komponente ist wie folgt aufgebaut.

((Listingkasten))

Listing 1: Aufbau eine Komponente in Angular 2.0

// dashboard.component.ts

import { Component, View } from 'angular2/angular2';

@Component({ selector: 'dashboard' })

@View({

template: `<p>{{ id }}</p>`

})

export default class DashboardComponent {

id: string = 'NG-Car 2015';

}

((Ende Listingingkasten))

Von Angular werden zunächst zwei Module *@Component()* und *@View()* importiert. Diese beiden Module sind im Speziellen TypeScript-Dekoratoren. Dekoratoren ermöglichen es Klassen zu

durch Meta-Angaben erweitern. *@Component()* spezifiziert, dass die Dashboard-Komponente über den *selector* <dashboard> im DOM des HTML-Dokuments eingesetzt wird.

Mit @View() definiert man das Template, das mit der Komponenten verknüpft ist. In diesem Beispiel wird das Feld *id*, aus der Klasse *DashboardComponent*, im Template gebunden und angezeigt. An dieser Stelle wird deutlich, was eine Komponente ist: Komponenten sind die neuen zentralen Bausteine von Angular 2.0. Sie übernehmen die Rolle von Direktiven und Controllern aus AngularJS.

Eine Komponente ist ein angereichertes Template, das im Browser zur Anzeige gebracht wird. Das Template verfügt über ein spezifisches Verhalten, das in Angular 2.0 durch TypeScript-Dekoratoren beschrieben wird.

Interpolation

Wie wird nun aus dem Ausdruck *{{ id }}* ein angezeigter Text im Browser?

Bereits in AngularJS 1.x konnten Daten mithilfe zweier geschweifter Klammern an ein HTML Template gebunden werden. Der Wert wurde mittels Interpolation ausgewertet und angezeigt.

Dieses Konzept bleibt in Angular 2.0 erhalten.

<p>{{ id }}</p>

Diese Schreibweise ist eine Vereinfachung der tatsächlichen Syntax. Denn bevor dieses Template im Browser ausgegeben wird, setzt Angular diesen Ausdruck in ein Property-Binding um. [[6]]

<p [text-content]="interpolate(['Gregor'], [name])"></p>

Komponenten miteinander verknüpfen

Um in dem Dashboard nun ein Auto abbilden zu können wird eine weitere Komponente benötigt.

((Listingkasten))

Listing 2: CarComponent

// car.component.ts

import { Component, View, Input } from 'angular2/angular2';

@Component({ selector: 'car' })

@View({

template: `<p>{{ id }}</p>`

})

export default class CarComponent {

@Input() id: string;

}

((Ende Listingingkasten))

Im ersten Schritt soll diese Komponente lediglich die zugewiesene Identifikationsnummer ausgeben. Die @Input()-Dekorator bietet die Möglichkeit, Werte an die *CarComponent* zu übergeben. Näheres wird im folgenden Abschnitt erläutert.

Nun kann die CarComponent im Dashboard referenziert und im Template verwendet werden.

((Listingkasten))

Listing 3: Integration der CarComponent in das Dashboard

// dashboard.component.ts

import { Component, View } from 'angular2/angular2';

import { CarComponent } from '../car/car.component';

@Component({ selector: 'dashboard' })

@View({

directives: [CarComponent],

template: `<car [id]="id"></car>`

})

export default class DashboardComponent {

id: string = 'NG-Car 2015';

}

((Ende Listingingkasten))

Im Wesentlichen wurden drei Anpassungen vorgenommen.

((Aufzählung))

Über ein weiteres import statement wird *CarComponent* geladen.

@View() wird durch die Eigenschaft `directives` ergänzt, damit *CarComponent* im Template verwendet werden kann.

Das Feld *id* wird and die gleichnamige Eigenschaft der *CarComponent* gebunden (Hierbei handelt es sich um ein Porperty-Binding).

((Ende Aufzählung))

So wurde über die Datenbindung die erste Interaktion zwischen zwei Komponenten realisiert.

Input- und Output-Properties

Input- und Output-Properties sind Eigenschaften die die API einer Angular-Komponente beschreiben. Über Inputs werden Informationen an eine Komponente übergeben. Mit Outputs kommuniziert die Komponente Änderungen nach außen.

Inputs werden durch `*Property-Bindings*` beschrieben. Outputs können über `*Event-Bindings*` abonniert werden.

Property-Bindings

Mit Properties werden einer Komponente Daten übermittelt.

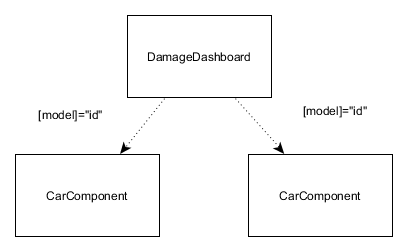


Abbildung 2 Daten an eine Komponente übergeben (Property-Binding)

Property-Bindgins zeichnen sich durch eckicke Klammern aus (`*[id]*`)

<car [id]="id"></car>

Anstatt eckiger Klammern können Property-Bindings auch mit der vollständigen Syntax `*bind-{property-name}="{expression}"*` beschrieben werden.

<car bind-id="id"></car>

Event-Bindings

Events bieten die Möglichkeit auf Veränderungen einer Komponente zu reagieren. Sie bieten einer Komponente die Möglichkeit mit ihrer Außenwelt zu kommunizieren.

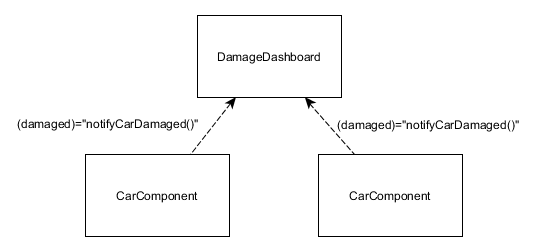


Abbildung 3 Zustandsänderungen innerhalb einer Komponente publizieren

Event-Bindings zeichnen sich durch runde Klammern aus (`*(damaged)*`). Sie triggern die Ausführung eines Statements.

<car (damaged)="report(damage)"></car>

Auch für diese Syntax existiert eine längere Syntax in der Form `*on-{event-name}="{statement}"*`

<car on-damaged="report(damage)"></car>

Um solch ein Event aus einer Komponente heraus zu erzeugen, wird der Dekorator `*@Output()*` verwendet. Das dazugehörige Property ist ein `*EventEmitter*`, der Ereignisse auslösen kann.

((Listingkasten))

Listing 4: Ereignisse innerhalb einer Komponente auslösen

// car.component.ts

import { EventEmitter } from 'angular2/angular2';

@Component({ /\* ... \*/ })

class CarComponent() {

@Input() id:string;

@Output() damaged:EventEmitter = new EventEmitter();

reportDamage() {

// Event auslösen

this.damaged.next(this.id);

}

}

((Ende Listingingkasten))

Neben der Verwendung runder Klammern, können Event-Bindings auch mit dem Ausdruck `*on-{Event-Name}="{callback()}"*` deklariert werden.

<car on-damaged="report(damage)"></car>

In der Dashboard Komponente muss lediglich eine Methode ergänzt werden, die nach dem Ausläsen des Events `*(damaged)*`, ausgeführt wird.

((Listingkasten))

Listing 5: Auf Ereignisse einer Komponente reagieren

// dashboard.component.ts

export default class DashboardComponent {

/\* ... \*/

notifyCarDamaged() {

this.totalDamages++;

}

}

((Ende Listingingkasten))

In diesem Fall wird im Dashboard die Anzahl der gemeldeten Schadensfälle zusammengezählt.

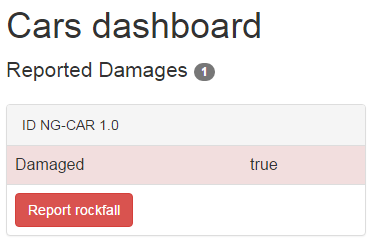


Abbildung 4 Anzeige der Anzahl aller erfassten Schäden

Two-Way Bindings mit `[(*ng-model)]*`

**ACHTUNG** Um die Direktive `[(ng-model)]` zu verwenden, muss vorher das Modul { FORM\_DIRECTIVES } importiert werden.

Aus Sicht einer Komponente werden mit Property-Bindings schreibende und den Event-Bindings lesende Operationen spezifiziert. Wie in AngularJS 1.x, ist es auch möglich Zwei-Wege-Bindungen (Two-Way-Bindings) zu realisieren. In der Template-Syntax von Angular 2.0 werden hierfür die Schreibweisen beider Binding-Arten kombiniert.

<input [(id)]="id">

Die eckigen Klammern legen fest, dass ein gegebener Wert an das <input>-Element gebunden wird. Die runden Klammern machen deutlich, dass Änderungen der Eigenschaft überwacht werden und diese mithilfe der Direktive `*ng-model*` in die Eigenschaft zurückschreiben werden.

Wie in den vorangehenden Beispielen gibt es auch hier eine alternative Schreibweise.

<input bindon-ng-model="id">

Die Zwei-Wege-Bindung lässt sich auch ohne `*ng-model*` realisieren. Das Markup wird so allerdings etwas komplexer.

<input

[value]="id"

(input)="id=$event.target.value">

Hierbei gibt `*$event*` Zugriff auf das auslösende Ereignis. Es ist ein natives Javascript-Event. Daher kann dessen API verwendete werden, um auf das betroffene Element zuzugreifen und dessen werden auszulesen (`*id=$event.target.value*`).

Template Referenzen

Innerhalb eines Templates können Referenzen auf HTML-Elemente, Komponenten und Datenbindungen erzeugt werden, um mit ihnen zu arbeiten.

<input #id type="text"/>

{{ id.value }}

Das Binding *{{ id.value }}* macht deutlich, dass die lokale Referenz das HTML-Element referenziert und nicht nur dessen Wert.

Anstatt der *#* können lokale Referenzen auch mit dem Prefix `*var-*` deklariert werden.

<input var-id type="text"/>

{{ id }}

Lokale Referenzen auf Komponenten unterscheiden syntaktisch nicht im Vergleich zu den HTML-ELementen. Zusätzlich können die Methoden der Komponente genutzt werden, um so mit ihr zu interagieren.

<car #car></car>

<button (click)="car.getTankCapicity()">Get tank capacity</button>

Lokale Referenzen können auch auf Objekte zeigen. Im folgenden Beispiel wird der Platzhalte `*#c*` genutzt, um für jedes Element der Liste `*cars*` die Komponente `*Car*` zu rendern.

<car \*ng-for="#c in cars" [model]="c">

Bei dem Stern (`\*`) vor der ng-for Direktive handelt es sich um eine Kurzschreibweise. Näheres erfahren Sie im nächsten Abschnitt.

\* und <template>

Direktiven wie `*ng-for*`,`*ng-if*` und `*ng-switch*` werden zusammen mit einem Stern (`*\**`) verwendet. Diese Direktiven werden strukturelle Direktiven (*Structural Directives*) genannt, da sie DOM-Elemente hinzufügen oder entfernen.

<div \*ng-if="totalDamages > 0">{{ totalDamages }}</div>

In diesem Beispiel wird das <div> Element nur in den DOM-Tree gezeichnet, wenn die Bedingung von `*ng-if*` wahr ist. Bei dem `\*` handelt es sich, um eine Kurzschreibweise, die das Schreiben des Templates vereinfachen soll. Sie wird als *Micro Syntax* bezeichnet, da Angular 2.0 diesen Ausdruck interpretiert und wieder in die bekannten Bindings umsetzt. Beispielsweise ist auch folgende Verwendung der ng-if Direktive zulässig.

<template [ng-if]="totalDamages > 0">

<div>{{ totalDamages }}</div>

</template>

Angular übersetzt die Micro Syntax in ein Property-Binding und umschließt das Template mit einem &lt;template&gt;-Tag. Dadurch entfällt der `\*` , vor dem `*ng-if*`.[[5]]

Zwei der (zahlreichen) neuen Klassen von Zend Framework 1.10.0 sind *Zend\_Filter\_Compress* und *Zend\_Filter\_Decompress*. Der Name ist Programm: damit ist es möglich Daten zu komprimieren und wieder zu entkomprimieren. Abhängig von der PHP-Installation und -Konfiguration unterstützt das Zend Framework eine ganze Reihe von Formaten und auch von Möglichkeiten des Packens beziehungsweise Entpackens. Beginnen wir zunächst mit den Datenformaten. Die folgenden werden unterstützt:

((Aufzählung))

Bz2

Gz

Lzf

Rar

Tar

Zip

((Ende Aufzählung))

Wundern Sie sich nicht über die ungewöhnlichen Schreibweisen (beispielsweise schreibt man in der Regel ZIP statt Zip) – die oben angegebenen Vorgaben werden so an späterer Stelle vom Zend Framework auch erwartet. Nicht all diese Formate funktionieren jedoch ohne weiteres Eingreifen; die zugehörige PHP-Erweiterung (etwa Bz2 für die BZip2-Unterstützung oder Zlib für die GZ-Unterstützung) muss vorhanden sein. Einzige Art von Ausnahme ist das TAR-Format, das nur bei Vorhandensein des PEAR-Pakets Archive\_Tar zur Verfügung steht.

Wie bereits zuvor erläutert sind die beiden zentralen Klassen *Zend\_Filter\_Compress* und *Zend\_Filter\_Decompress*. Beginnen wir mit dem Komprimieren von Daten. Diese Daten können Dateien oder auch einfach Strings sein. Das in Listing 1 gezeigte HTML-UI stellt die Grundlage für ein kleines Beispielskript dar.

((Listingkasten))

Listing 1: HTML-UI

<form method="get" action="">

<input type="text" name="q" /><br />

<input type="radio" name="typ" value="Bz2" />BZip2

<input type="radio" name="typ" value="Gz" />GZip

<input type="radio" name="typ" value="Lzf" />Lzf

<input type="radio" name="typ" value="Rar" />Rar

<input type="radio" name="typ" value="Tar" />Tar

<input type="radio" name="typ" value="Zip" />ZIP<br />

<input type="submit" value="Komprimieren" />

</form>

((Ende Listingingkasten))

Auf Mausklick soll der eingegebene Text im gewählten Format komprimiert werden – wobei Rar und Tar nur der Vollständigkeit halber aufgeführt sind, denn sie unterstützen keine Zeichenketten als Eingabe, lediglich Dateien und Verzeichnisse (dazu später mehr).

Das eigentliche Komprimieren wiederum ist schnell geschehen. Zunächst muss die Klasse *Zend\_Filter\_Compress* instanziiert werden; als Argument geben Sie das Format gemäß obiger Liste an.

$filter = new Zend\_Filter\_Compress('Lzf');

Im nächsten Schritt wird die *filter()*-Methode aufgerufen. Ergebnis: der komprimierte Text.

$filter->filter($q)

Wie in Bild 2 zu sehen, lohnt sich das Vorgehen natürlich nur ab einer gewissen Textmenge, damit die komprimierten Informationen auch wirklich weniger Speicherplatz benötigen als die Quelldaten. Listing 2 zeigt den relevanten Code für die HTML-Oberfläche.

((Listingkasten))

Listing 2: HTML-Oberfläche

$typen = array('Bz2', 'Gz', 'Lzf', 'Rar', 'Tar', 'Zip');

if (isset($\_GET['q']) && is\_string($\_GET['q']) &&

isset($\_GET['typ']) && in\_array($\_GET['typ'], $typen)) {

require 'Zend/Filter/Compress.php';

$q = $\_GET['q'];

$filter = new Zend\_Filter\_Compress($\_GET['typ']);

echo 'Ergebnis: ' . htmlspecialchars($filter->filter($q));

}

((Ende Listingingkasten))

*Zend\_Filter\_Compress* kann aber auch Dateien erzeugen, die dann die komprimierten Informationen enthalten. Sie müssen dazu nur noch ein paar zusätzliche Daten angeben, wenn Sie mit der Klasse arbeiten – und zwar in Form eines Arrays. Zwei Werte sollten Sie angeben:

((Aufzählung))

*adapter*: Das Dateiformat

*options*: Ein Unter-Array mit weiteren Informationen wie dem Dateinamen des Archivs (zugehöriger Schlüssel: *archive*)

((Ende Aufzählung))

Hier ein Beispiel:

$optionen = array(

'adapter' => 'Lzf',

'options' => array(

'archive' => '/pfad/zu/datei.lzf'));

Das Beispiel auf der PHP-Journal-Website arbeitet mit derselben Oberfläche wie das vorangegangene Listing. Auf Basis des Komprimierungstypen wird die Dateiendung ermittelt und dann die Datei *datei.xyz* erzeugt – *xyz* steht für das Dateiformat, beispielsweise *zip*. Beachten Sie unbedingt, dass der Code lediglich zu Erklärungszwecken dient. Sollte es die angegebene Datei bereits geben, wird sie ohne Rückfrage überschrieben. Und PHP benötigt natürlich auch Schreibrechte für den Zielort (Bild 3).

Genug der Vorrede, Listing 3 zeigt den relevanten zugehörigen Code.

((Listingkasten))

Listing 3: Komprimierung

require 'Zend/Filter/Compress.php';

$q = $\_GET['q'];

$dateiname = 'datei.' . strtolower($\_GET['typ']);

$optionen = array(

'adapter' => $\_GET['typ'],

'options' => array('archive' => $dateiname));

$filter = new Zend\_Filter\_Compress($optionen);

$filter->filter($q);

echo "<a href=\"$dateiname\">Zur Datei</a>";

((Ende Listingingkasten))

Die umgekehrte Richtung – Daten entpacken statt packen – geht natürlich auch. Zuständige Klasse ist wie oben bereits erwähnt *Zend\_Filter\_Decompress*. Als Eingabeparameter für die *filter()*-Methode kommen diesmal die komprimierten Daten bzw. der zugehörige Dateiname zum Einsatz:

$filter->filter('datei.zip');

Für den Klassenkonstruktor jedoch benötigen wir wieder ein paar Optionen, die in Form eines Arrays übergeben werden. Ein alter Bekannter, *adapter*, gibt erneut an was für ein Komprimierungsformat vorliegt. Im Konfigurationsarray *options* instruiert der Wert *target* das Zend Framework, wohin die Datei entpackt werden soll. Die zuvor schon vorgebrachten Warnhinweise gelten auch hier: PHP benötigt Schreibrechte in das angegebene Verzeichnis, und man sollte nicht wie im Beispiel einfach Dateien ohne zusätzliche Prüfungen entpacken.

Für unsere Zwecke reicht aber ein vereinfachtes Szenario bereits aus. Der Nutzer kann per HTML-Datei-Upload ein Archiv an den Server übertragen sowie das zugehörige Dateiformat auswählen. Letzteres könnte man zwar auch aus der Dateiendung versuchen zu ermitteln, aber so vereinfacht sich das Testen ein wenig (Listing 4).

((Listingkasten))

Listing 4: Datei-Upload

<form method="post" action="" enctype="multipart/form-data">

<input type="file" name="datei" /><br />

<input type="radio" name="typ" value="Bz2" />BZip2

<input type="radio" name="typ" value="Gz" />GZip

...

<input type="submit" value="Dekomprimieren" />

</form>

((Ende Listingingkasten))

Der serverseitige Code (Listing 5) greift auf die übertragene Datei zu und entpackt sie mithilfe des Filters.

((Listingkasten))

Listing 5: Entpacken auf dem Server

require 'Zend/Filter/Decompress.php';

$dateiname = $\_FILES['datei']['tmp\_name'];

$optionen = array(

'adapter' => $\_POST['typ'],

'options' => array('target' => './entpackt'));

$filter = new Zend\_Filter\_Decompress($optionen);

$filter->filter($dateiname);

echo '<p>Archiv entpackt.</p><p>';

((Ende Listingingkasten))

Sie können dann im Dateisystem des Servers überprüfen ob der Dekomprimierungsvorgang erfolgreich war. Zusätzlich gibt der Code in Listing 5 noch den Inhalt des Zielverzeichnisses an. Bedenken Sie aber dass bei mehrmaligem Entpacken in dasselbe Verzeichnis sowohl noch Altdateien vorhanden oder gar überschrieben worden sein könnten (Bild 4).

((Listingkasten))

Listing 5: Check des Dateisystems

$dir = dir('./entpackt');

while (($datei = $dir->read()) !== false) {

if ($datei != '.' && $datei != '..') {

echo htmlspecialchars($datei) . '<br />';

}

}

$dir->close();

echo '</p>';

((Ende Listingingkasten))

Ver-/Entschlüsseln

Die ursprüngliche Intention der Filter-Klasse von Zend Framework war es, Sicherheitsfeatures anzubieten. Wir gerade gesehen ist die Funktionalität mittlerweile allgemeiner einsetzbar. Einige der Klassen unterhalb von *Zend\_Filter* tangieren jedoch durchaus sicherheitsrelevante Themen. Zwei davon stellen die folgenden Codebeispiele vor: *Zend\_Filter\_Encrypt* und *Zend\_Filter\_Decrypt*. Wie der Name bereits andeutet können damit Dateien ver- und wieder entschlüsselt werden. Die Implementierung der Klasse basiert dabei im Wesentlichen auf anderen PHP-Erweiterungen oder –Funktionalitäten. Prinzipiell gibt es zwei technische Ansätze für die Datenverschlüsselung bzw. –entschlüsselung, die von *Zend\_Filter\_En*/*Decrypt* unterstützt werden: Die *mcrypt*-Erweiterung von PHP und die *OpenSSL*-Erweiterung von PHP. Im Folgenden werfen wir einen Blick auf erstere Option. Die HTML-Oberfläche unseres Beispiels ermöglicht es einen Text zu verschlüsseln:

<form method="post" action="">

Text: <input type="text" name="text" /><br />

Schlüssel: <input type="text" name="key" /><br />

<input type="submit" value="Verschlüsseln" />

</form>

Mcrypt benötigt drei Informationen für die Verschlüsselung: Zunächst einmal die Zeichenkette, die zu verschlüsseln ist. Zweitens benötigen wir ein geheimen Schlüssel, und drittens einen Vektor der bei der Ver- und Entschlüsselung zum Einsatz kommt. Sie können im Code den Vektor selbst auch angeben (die Methode heißt *setVector()*), aber einfacher ist es wenn Sie das Zend Framework selbst einen zufälligen Vektor bestimmen lassen. Wichtig dabei: die Methode *getVector()* liefert diesen Vektor zurück. Ohne diesen Vektor kann die Verschlüsselung nicht mehr rückgängig gemacht werden.

Im Codeschnipsel in Listing 6 sehen Sie die Funktionsweise des Filters, die ganz ähnlich zu den anderen Filter-Klassen ist. Im Konstruktor werden per Arrays ein paar Informationen angegeben, beispielsweise der geheime Schlüssel. Auch die zu verwendende Verschlüsselungsmethode kann dort angegeben werden; wir setzen alternativ auf die Methode *setAdapter()*. Anschließend gibt der Code nicht nur das Ergebnis der Verschlüsselung aus, sondern auch den verwendeten Vektor – wir wollen ja später noch einmal an unsere Daten heran.

((Listingkasten))

Listing 6: Funktionsweise des Filters

if (isset($\_POST['text']) && is\_string($\_POST['text']) &&

isset($\_POST['key']) && is\_string($\_POST['key'])) {

require 'Zend/Filter/Encrypt.php';

$text = $\_POST['text'];

$filter = new Zend\_Filter\_Encrypt(array('key' => $\_POST['key']));

$filter->setAdapter('mcrypt');

$vector = $filter->getVector();

echo 'Ergebnis: ' . $filter->filter($text) . '<br />';

echo 'Verwendeter Vektor: ' . $vector;

}

((Ende Listingingkasten))

Die HTML-Oberfläche in Listing 7 wird uns dabei behilflich sein, wieder die ursprünglichen Daten zu erhalten: ein Benutzer kann die verschlüsselte Zeichenketten, den verwendeten Schlüssel und natürlich den Vektor eingeben:

((Listingkasten))

Listing 7: Formular für Abfrage

<form method="post" action="">

Verschlüsselter Text: <input type="text" name="text" value="<?php

echo htmlspecialchars($text);

?>" /><br />

Schlüssel: <input type="text" name="key" /><br />

Vektor: <input type="text" name="vector" value="<?php

echo htmlspecialchars($vector);

?>" /><br />

<input type="submit" value="Entschlüsseln" />

</form>

((Ende Listingingkasten))

Die zugehörige Klasse im Zend Framework heißt *Zend\_Filter\_Decrypt*. Das weitere Prozedere kennen Sie jetzt ja: Klasse instanziieren, Adapter angeben, Vektor angeben, und filtern (Listing 7).

((Listingkasten))

Listing 7: Entschlüsseln

require 'Zend/Filter/Decrypt.php';

$text = $\_POST['text'];

$vector = $\_POST['vector'];

$filter = new Zend\_Filter\_Decrypt(array('key' => $\_POST['key']));

$filter->setAdapter('mcrypt');

$filter->setVector($vector);

echo 'Ergebnis: ' . htmlspecialchars($filter->filter($text));

((Ende Listingingkasten))

Bei Verwendung von OpenSSL ändern sich einige der Rahmenparameter. Statt Schlüsseln und Vektoren benötigen Sie Schlüsseldateien und eine Passphrase. Letzten Endes läuft es aber wie zuvor ab: Sie geben ein paar Daten und Informationen an und rufen abschließend die Methode *filter()* auf.

Fazit

Das Zend Framework liefert über zwei Dutzend verschiedene Filterklassen mit – und dabei sind Einzelfilter wie etwa die für die verschiedenen Komprimierungsformate gar nicht erst mitgezählt. Der Ordner *Filter* im Zend Framework enthält über 50 Dateien.

((Autorenkasten))

((autorenbild.jpg))

Autor

**Christian Wenz** ist Autor, Trainer und Berater mit den Schwerpunkten Webentwicklung und Webtechnologien. Er bloggt sporadisch unter

www.hauser-wenz.de/blog/

((Ende Autorenkasten))

((Kasten))

Links zum Thema

Download des Zend Framework und zugehöriger Dokumentation

framework.zend.com/

Dokumentation zu *Zend\_Filter*

framework.zend.com/manual/en/zend.filter.html

PHP-Handbuchseite zur mcrypt-Erweiterung

php.net/mycrpt

PHP-Handbuchseite zur OpenSSL-Erweiterung

php.net/openssl

((Ende Kasten))

((Bildunterschriften))

((0406\_ framework10.png))

**Brandneu:** Version 1.10.0 des Zend Framework erschien Ende Januar 2010 (Bild 1)

((0406\_bz-string.png))

**Fettkur:** Bei kleinen Daten ist das Komprimierungsergebnis zu groß Bild 2

((0406\_bz-file.png))

**Kleindatei:** Das Archiv enthält tatsächlich die eingegebenen Informationen (Bild 3)

((0406\_fileupload.png))

**Ausgepackt:** Die Archivinhalte wurden extrahiert (Bild 4)