Vimerito 2 MVC

Für Version 0.6.1 Beta

Version des Handbuches 0.4.1

Autor: Benjamin Werner

Inhaltsverzeichnis

1. Einführung

Vimerito bietet dir die Möglichkeit mit geringem Aufwand maximale Erfolge zu erzielen. Egal ob es sich um eine kleine Webseite oder um eine komplexe Webanwendung handelt. Die großen Vorteile von Vimerito liegen in seiner hohen Geschwindigkeit, mit der eine komplette Seite abgearbeitet wird und der einfachen Handhabung. In der Programmierung mit Vimerito ist vieles ein „Kann“, das Wenigste aber ein „Muss“. Was ich damit sagen will ist, dass es in Vimerito möglich ist mit einfachen Mitteln beachtliche Erfolge zu erzielen, wobei Vimerito einen großen Teil der Arbeit abnimmt. Wer sich dadurch in den Freiheiten eingeengt fühlt, kann aber auch tiefer in Vimerito einsteigen und Funktionen nach eigenen Wünschen anpassen.

2. Konfigurieren von Vimerito

2.1 Konfigurieren der Anwendung

Vimerito wurde so konzipiert, dass es einfach und schnell konfiguriert werden kann. Für die minimale Konfiguration sind lediglich 4 Werte von Nöten:

- Der Seiten-URL
- Die Seitenadresse
- Der Standartcontroller
- Die Standartmethode

In der Datei

`DOCUMENT-ROOT/application/configuration/applicationConfiguration.php`

wird ein Array Namens `$__cachedApplicationConfiguration` initialisiert.
Die folgende Tabelle gibt dir eine Übersicht über die einzelnen Einstellungsmöglichkeiten von Vimerito.

`'pageUrl'`	Gibt den URL zu deiner Internetseite an. D.h. http://www.deine-seite.com
`'pageAdress'`	Gibt die Adresse zu deiner Seite an. D.h. www.deine-seite.com
`'routingParamtersSeperator'`	Entspricht einem Zeichen, dass in einem URL einen Parameter von seinem Wert trennt. Näheres findest du in der Routing-Regelung.
`'routeControllerOnDefault'`	Gibt an, welcher Controller aufgerufen werden soll, wenn im URL keiner an gegeben wurde.
`'routeMethodOnDefault'`	Gibt an, welche Methode des Standartcontrollers aufgerufen werden soll, wenn keine im URL an gegeben wurde. Wenn dieser Wert leer bleibt, wird die allgemeine Standartmethode auf gerufen. Näheres dazu findest du im Abschnitt „Controller“
`'javaScriptMode'`	Dieser Wert ist entweder `true` oder `false`. Wenn er auf `true` gesetzt wurde, wird in jede Seite automatisch das JavaScript-Framework „jQuery“ eingebunden.

`'automaticAuthentication'`	Schaltet die automatische Benutzerauthentifizierung ein. Dieser Wert ist standardmäßig `false`. Schaue für nähere Informationen in den Abschnitt „Authentifizierung“.
`'applications'`	Dieser Wert enthält ein Array mit Namen von Applikationen und deren Pfade. Konsultiere den Abschnitt „Applications“.

2.2 Konfigurieren der Datenbank

Um eine Datenbank mit Vimerito nutzen zu können, muss die Konfigurationsdatei `DOCUMENT-ROOT/application/configuration/databaseConfiguration.php` angepasst werden. In dieser Datei wird das Array `$DatabaseConfiguration` erzeugt.

Folgende Werte sind auszufüllen:

`'server'`	Name des Servers auf dem die Datenbank liegt. In den meisten Fällen ist dies `'localhost'`.
`'database_name'`	Name der Datenbank.
`'username'`	Benutzername für die Datenbank.
`'user_password'`	Passwort für die Datenbank.
`'prefix'`	Es kann ein Prefix erstellt werden. Somit können mehrere Versionen einer Datenbank

Aktuell unterstützt Vimerito nur Mysql-Datenbanken.

2.3 Starten einer Anwendung

Um eine Anwendung aus führen zu können muss eine Index-Datei angelegt werden. Bei den meisten Webserver wird die Datei `index.php` auf gerufen, sobald ein Benutzer auf deine Internetseite zugreift. In manchen Fällen, und das ist wirklich selten, wird die Datei `home.php` auf gerufen. Diese Datei muss konfiguriert werden. Grundsätzlich bedarf es dabei zum Starten einer Anwendung nur 3 Zeilen PHP-Code. Um aber Vimerito 2 im vollen nutzen zu können muss sich die Datei .htaccess im Wurzelverzeichnis befinden. In dieser Datei wird festgelegt, wie Vimerito mit übergebenen Werten um geht. Außerdem sorgt diese Datei dafür, dass suchmaschinenfreundliche URL's benutzt werden können.
Was unterscheidet normale URL's von suchmaschinenfreundlichen?
Im Grunde genommen hängt das mit dem Aufbau eines URL's zusammen. Eine normaler URL hat in aller Regel die Form:
`http://www.meine-seite.de?seite=home&funktion=init&parameter1=wert1&parameter2=wert2`
So etwas mögen Suchmaschinen nicht. Viel zu lang und unleserlich.
Der Beispiel-URL in Vimerito 2 sieht wie folgt aus:
`http://www.meine-seite.de/home/init/parameter1-wert1/parameter2-wert2.html`
Zuerst fällt auf, dass der URL in Vimerito 2 wesentlich kürzer und besser zu lesen ist. Gleichzeitig ist er einprägsamer, was Benutzern die Möglichkeit gibt Unterseiten deiner Website direkt in die Adresszeile ein zugeben, obgleich die meisten Benutzer dazu viel zu faul sind.

Aber noch etwas unterscheidet das obere Beispiel von dem unteren. Das untere macht den Eindruck als würde eine normale HTML-Seite in einem Unterverzeichnis auf gerufen werden. Dies ist aber nicht der Fall. Vimerito 2 verarbeitet den URL und ruft entsprechend bestimmter Regeln die entsprechenden PHP-Skripte auf.
Desweiteren sind solche URL's ein Vorteil bei der Suchmaschinenplatzierung.
Die Datei .htaccess enthält folgdenen Code:

```
RewriteEngine on
RewriteRule ^(.*).html ./index.php\?param=$1 [L]
```

In die Datei `index.php` muss Vimerito 2 zunächst eingebunden werden. Danach wird Vimerito 2 initialisiert und im letzten Schritt wird die Anwendung gestartet.

```
<?
   require "system/classes/Vimerito.class.php";
   Vimerito::initApplication();
   Vimerito::runApplication();
?>
```

Manchmal ist es nötig, um Fehlern im Code auf die Spur zu kommen, sich alle Benachrichtigungen die PHP in Fehlerfällen ausgibt, sich an zeigen zu lassen. Im Standard unterdrückt Vimerito 2 alle Fehlerausgaben bis auf schwerwiegende Fehler. Möchte man dies ändern, muss bevor die Applikation gestartet wird PHP mitteilen, was ausgegeben werden soll. Soll alles ausgegeben werden könnte man den Code wie folgt umgestalten:

```
 <?
   require "system/classes/Vimerito.class.php";
   Vimerito::initApplication();
   error_reporting(E_ALL);
   Vimerito::runApplication();
?>
```

3. Die Komponenten

Vimerito ist vom grundsätzlichen Aufbau ein klassisches MVC. Jedes MVC unterteilt sich in 3 grundlegende Komponenten.

- Controller
- View
- Model

Jeder Komponente kommt eine spezielle Aufgabe bei der Abarbeitung des Programmcodes zu. Die Aufgaben sollten beim Programmieren strikt getrennt gehalten und nicht vermischt werden.
Jede Seite besteht mindestens aus einer Controller- und einer View-Komponente.

3.1 Allgemeine Namenskonventionen

Damit Vimerito 2 die einzelnen Komponenten finden und automatisch laden kann, gibt es bestimmte Regeln für die Benennung von Klassen und Dateien.
Keine Klasse darf den gleichen Namen, wie eine Systemklasse haben.
Alles was vor einem „_“ steht wird als Ordner interpretiert, ausgehend vom Standartverzeichnis der entsprechenden Komponente.
Controller liegen beispielsweise im Verzeichnis `DOCUMENT-ROOT/application/controllers`.
Eine Controller-Klasse, die den Namen `user_profile` hat, wird in der Datei `DOCUMENT-ROOT/application/controllers/user/profileController.class.php` gesucht.

3.2 Das Model

Die Model-Komponente hat die Aufgabe Daten zu sammeln und bereit zu stellen. Dabei ist es gleich, ob die Daten aus einer Datenbank oder einem Formular stammen. In Vimerito gibt es zwei Arten von Models.

1. Der ActivRecorder
2. Das FormModel

3.2.1 Der ActivRecorder

Der ActivRecorder ist als eine Schnittstelle zu einer Datenbanktabelle zu sehen. In Vimerito gibt es den Vorteil das ein ActivRecorder weitest gehend konfigurationsfrei ist. Jeder ActivRecorder leitet sich von der Klasse `VActivRecorder` ab. Er sollte im Ordner

`DOCUMENT-ROOT/application/models` liegen. Der Name eines solchen Modeldatei sollte folgende Konventionen erfüllen:

1. Groß- und Kleinschreibung wird beachtet
2. Der Dateiname sollte folgendes Format haben: {Klassenname}{„Model“}.Endung

   Ein Beispiel: UsertabelleModel.class.php
3. Der Klassenname muss dem Tabellennamen entsprechen, auf welche dieses Model zeigen soll.

Die Programmierung ist sehr einfach gehalten. Die Modelklasse muss sich von der Klasse `VActivRecorder` ableiten. Zusätzlich muss nur ein Konstruktor geschrieben werden, welcher den Konstruktor der Vaterklasse aufruft. Ein Beispiel:

```
class Meinetabelle extends VActivRecorder{
    public function __construct(){
     parent::__construct();
     }
}
```

Dieses Beispiel erzeugt ein Model welches auf die Tabelle „Meinetabelle“ verweist. Wird der Vaterkonstruktor parameterlos oder mit `true` aufgerufen, wird die Tabelle untersucht und ihr Aufbau für die Klasse übernommen. Dabei wird der Aufbau dieser Tabelle gecached, also in einer Datei abgelegt. Wird dieses Model also ein zweites Mal aufgerufen, wird nicht mehr die Tabelle untersucht, sondern nur die gecachte Tabellenkonfiguration übernommen. Ziel ist es, so wenig Mysql-Befehle an die Datenbank zu schicken, wie möglich. Dieses Verhalten von Vimerito macht jedoch nur Sinn, wenn sich der Aufbau der Tabelle nicht mehr ändert. Vimerito bietet eine Möglichkeit dieses Verhalten zu umgehen. Soll der Aufbau der Tabelle aus dem letzten Beispiel nicht gecacht werden, muss die Modelklasse wie folgt aussehen:

```
class Meinetabelle extends VActivRecorder{
    public function __construct(){
     parent::__construct(false);
     $this->analyseDatabase(false);
     }
```

```
}
```

Der Vaterkonstruktor wird also mit dem Parameter `false` aufgerufen. Das bedeutet, dass die Tabelle nicht automatisch untersucht wird. Im nächsten Schritt wird nun die Methode `analyseDatabase` mit einem `false` aufgerufen. Das sagt dem ActivRecorder, dass in jedem Fall die Tabelle untersucht werden muss und das Ergebnis nicht gecacht wird. Wird diese Methode wiederum ohne Parameter oder mit einem `true` aufgerufen, entspricht dies dem erstem Beispiel. Das Model benötigt nach dieser Initialisierung keine zusätzlichen Methoden um auf die Datenbanktabelle zu zugreifen.

3.2.2 Verwendung des ActivRecorders

Der ActivRecorder bietet verschiedene Möglichkeiten Daten aus einer Datenbanktabelle zu filtern.

`findByPK` ist eine gängige Methode. Es wird also eine Zeile aus der Tabelle gesucht, wobei der Primärschlüssel verglichen werden soll.

Greifen wir das Beispiel unserer Usertabelle auf. Stellen wir uns vor, die Tabelle hat folgenden Aufbau:

ID	BenutzerID und Primärschlüssel
Name	Benutzername
Email	Emailadresse
PW	Passwort des Benutzers

Und die Tabelle enthält folgende Zeilen:

ID	Name	Email	PW
1	Peter	peter@email.com	123456
2	Paula	paula@email.com	78910
3	Olaf	olaf@olaf-tv.de	olaftv

Wenn du nun den Benutzer mit der ID 2 aus der Datenbank suchen möchtest, sieht der Befehl wie folgt aus:

```
$user = new Usertabelle;
$user->findByPK('2');
echo $user->Name;
//gibt „Paula“ aus
```

Diese 3 Zeilen Programmcode sagen einiges über das Verhalten des ActivRecorders aus.

1. Wenn du deine Modelklasse definiert hast, kannst du von jeder Stelle deines Anwendung darauf zu greifen. Das Hauptscript sucht automatisch die richtige Modeldatei und bindet diese ein.
2. Du kannst auf die Spalten direkt zugreifen, indem du sie als Attribute deiner Modelklasse aufrufst.
3. Du kannst immer nur auf das aktuelle Ergebnis einer Datenbankabfrage zugreifen.

Es wurden außerdem automatisch alle Spalten der Tabelle übernommen. Das ist nicht immer sinnvoll und führt zu hoher Datenbankbelastung. Wenn wir uns sicher sind, dass wir wirklich nur den Namen des Benutzers mit der ID 2 benötigen, können wir das letzte Beispiel, wie folgt umgestalten.

```
$user = new Usertabelle;
$user->findByPK('2', array('Name'));
echo $user->Name;
//gibt „Paula“ aus
echo $user->Email;
//gibt eine leere Zeichenkette aus
```

Der zweite Parameter ist ein Array in dem du mit einem Komma getrennt die Spalten angeben kannst, deren Werte geladen werden sollen.

Das Suchen mehrerer Ergebnisse ist natürlich auch möglich. Für solche Aufgaben gibt es den Befehl:
`findWhere`.

```
$user = new Usertabelle;
$user->findWhere(array(
    array('Name', '', '<>'),
    array('Email', '', '<>')
));
echo $user->Name;
//gibt „Peter“ aus
echo $user->Email;
//gibt peter@email.com aus
$user->next();
echo $user->Name;
//gibt „Paula“ aus
echo $user->Email;
//gibt paula@email.com aus
```

Das obige Beispiel sucht alle Benutzer, die einen Benutzernamen und eine Emailadresse haben, die also nicht leer („“) sind. Die Kriterien, nach denen gesucht werden soll, werden bei der findWhere-Methode als Array übergeben, das Arrays enhält. Die inneren Arrays stehen jeweils für ein Kriterium. Die obige Abfrage als Mysql-Befehl steht für

```
SELECT * FROM Usertabelle WHERE `Name`<> '' AND `Email`<>''
```

Wenn wir diese Abfrage so modifizieren wollen, dass entweder der Benutzername oder die Emailadresse nicht leer („“) sein darf, passen wir den Quelltext wie folgt an:

```
$user->findWhere(array(
    array('Name', '', '<>', 'OR'),
    array('Email', '', '<>')
));
```

Der resultierende Mysql-Befehl sieht nun so aus:

```
SELECT * FROM Usertabelle WHERE `Name`<> '' OR `Email`<>''
```

Der Aufbau eines solchen Kriteriums ist

1.	Name der Spalte
2.	Wert auf den geprüft werden soll

3.	Vergleichsoperator. Wenn dieser nicht angegeben ist, dann ist dieser „=“. Mögliche Vergleichsoperatoren sind: • = • <> • > • < • >= • <=
4.	Verknüpfungsoperatur zum nächsten Kriterium. Wenn dieser nicht angegeben ist, wird dafür „AND“ (logisches UND) eingesetzt. Wenn kein weiteres Kriterium folgt, bleibt dieser leer.

Doch bei genauerer Betrachtung des letzten Beispieles könnte dir ein neuer Befehl aufgefallen sein.

```
$user->next();
```

Nun wird die genaue Funktionsweise des ActivRecorders klar. Du erhältst die Möglichkeit alle Ergebnisse deiner Abfrage zu durchlaufen, aber auch auf ein schon gesehenes Ergebnis zu springen. Folgende Befehle hält der ActivRecorder bereit:

`first()`	Springt auf das erste Ergebnis der Ergebnismenge
`last()`	Springt zum letzten Ergebnis der Ergebnismenge
`next()`	Springt zum nächsten Ergebnis
`prev()`	Springt zu vorherigen Ergebnis
`isLast()`	Gibt `true` zurück, wenn das aktuelle Ergebnis das letzte ist. Andernfalls `false`.

Auch in im Falle des letzten Beispieles kann an gegeben werden, welche Spalten aus der Ergebnismenge gefiltert werden sollen. Außerdem können wir unsere Ergebnisse sortieren lassen und ein Limit festlegen.

```
$user->findWhere(
    array(
     array('Name', '', '<>', 'OR'),
     array('Email', '', '<>')
    ),
    array('Name', 'Email'),
    array('Name'=>'DESC',
    array(1)
);
```

Ein weiterer praktischer Befehl ist `findLast()`. Mit ihm kann man sich die letzten Einträge einer Tabelle ausgeben lassen. Auch hier kann wieder festgelegt werden, welche Spalten in das Ergebnis mit aufgenommen werden sollen, wie viele „letzte“ Zeilen, wie die Ergebnisse geordnet werden sollen und natürlich eine WHERE-Klausel.

```
$model->findLast(
    array('spalte1', 'spalte2'),
    array(10), array('spalte1' => 'DESC'),
```

```
    array(
    array('spalte2', '2', '=')
);
```

Diese Abfrage sucht in einer Tabelle die 10 letzten Einträge und gibt uns die Spalten „Spalte1“ und „Spalte2“ aus. Außerdem werden unsere Ergbenisse absteigend nach „Spalte1“ geordnet und „Spalte2“ muss den Wert „2“ besitzen.

3.2.3 Komplexe Datenbankabfragen

Zwar wird der ActivRecorder stetig ausgebaut, komplexere Mysql-Abfragen lassen auf die eben beschriebene Art aber nicht realisieren. Deshalb ist es möglich selbst definierte Abfragen an die Datenbank zu senden. Das Schlüsselwort heißt `send()`. Mit diesem Befehl kannst du komplette SQL-Queries an die Datenbank senden. Aber Achtung: Der ActivRecorder weiß bei diesem Befehl nicht, welche Tabelle deiner Datenbank gemeint ist, das soll heißen: Die betreffenden Tabellen sind in dem Query unbedingt mit an zu geben:

```
$model->send('SELECT * FROM tabelle');
```

Als zweiten optionalen Parameter kann dem Befehl ein TRUE oder FALSE übergeben werden. Dieser bestimmt, ob ein mögliches Ergebnis der Abfrage, dem Recordset übergeben werden soll, die Ergebnisse also, wie zuvor angesprochen werden sollen.

3.2.4 Mehrere Datenbanken

Nicht immer ist es möglich oder auch von Nutzen, vor allem bei größeren Projekten, das alle Tabellen in einer Datenbank liegen. Seit Version 0.6 gibt es hierfür Abhilfe. Für diesen Zweck wurde das Attribut `$__databaseConfiguration` eingeführt. Wird diese vor dem Aufruf des Vaterkonstruktors in einem Model angepasst, so kann die betreffende Tabelle in einer anderen Datenbank liegen. Die Einstellung dieses Arrays erfolgt äquivalent zur allgemeinen Datenbankkonfiguration.

3.2.5 Suche eines Ergebnisses

Vimerito 2 bietet die Möglichkeit aus einer Ergebnismenge ein bestimmtes Ergebnis heraus zu suchen. Mit der Methode `getResultByValue` kann der Wert einer Datenbankspalte abgefragt und mit einem übergebenen Wert verglichen werden.

```
$model->getResultByValue('spaltenName', 'wert1');
```

In diesem Beispiel überprüft der ActivRecorder ob in einem Ergebnis die Spalte „SpaltenName“ den Wert „wert1“ hat. Ist dies der Fall wird nur diese Ergebnisreihe als Objekt zurück geliefert. Wird kein passendes Ergebnis gefunden liefert diese Methode `Null` zurück.

3.2.6 Das FormModel

Das FormModel wird benutzt, um Formulardaten zu sammeln und bereit zu stellen. Dabei übernimmt es die Prüfung der Eingaben und das heraus filtern der Daten aus der Superglobalen `$_POST` nach Absenden eines Formulars. Ein FormModel sollte im Ordner

`DOCUMENT-ROOT/application/forms` liegen. Der Name eines solchen Formulars sollte folgende Konventionen erfüllen:

1. Groß- und Kleinschreibung wird beachtet

2. Der Dateiname sollte folgendes Format haben: {Klassenname}{„Form“}.Endung

   Ein Beispiel: RegisterForm.class.php

Jedes FormModel leitet sich von der Klasse `VFormModel` ab. Damit Vimerito weiß welche Felder in diesem Formular vorhanden sind, müssen diese zunächst im Konstruktor registriert werden. Für diese Aufgabe existiert das Array `Fields`. Eine Konfiguration eines Registrierungsformulars könnte wie folgt aus sehen:

```
class Register extends VFormModel{
    public function __construct(){
    $this->Fields = Array(
         'email'  =>  array(
                  'type'      =>  'input',
                  'size'      =>  '35',
                  'label'     =>  'Emailadresse'
                  ),
             'pw1'  =>  array(
                  'type'      =>  'password',
                       'size'      =>  '35',
                       'label'     =>  'Passwort'
             ),
             'pw2'  =>  array(
                       'type'      =>  'password',
                       'size'      =>  '35',
                  'label'     =>  'Passwort wiederholen'
                  ),
                  'submited'      =>  array(
                       'type'      =>  'submit',
                       'size'      =>  '35',
                       'value'     =>  'Registrieren'
             )
    );
    parent::__construct();
    }
}
```

Mit dieser Einstellung werden 3 Eingabefelder (`'email'`, `'pw1'`, `'pw2'`) und der Absendebutton (`'submitted'`) erzeugt. Diese Felder sind nun registriert. Ab diesen Zeitpunkt können sie benutzt werden.

Die einzelnen Felder sind assoziative Arrays. Für jedes Feld muss ein Typ festgelegt werden. Wie in HTML existieren die Typen:

`input`	Ein einfaches Eingabefeld der Länge `size`.
`password`	Ein Eingabefeld für die verdeckte Eingabe von Passwörtern der Länge `size`.
`number`	Ein Eingabefeld speziell für Zahlen der Länge `size`.
`listbox (Select)`	Auswahlliste bei der `size`-viele Elemente angzeigt werden.
`checkbox`	Auswahlkasten mit den Zuständen `true` und `false`.
`button`	Ein normaler Knopf.

`submit`	Ein Knopf mit dem automatisch das Formular abgeschickt wird.

Die Einstellungsmöglichkeiten orientieren sich an denen von HTML. Jedoch wurden noch ein paar weitere Angaben optional hinzugefügt:

Label	Die Beschriftung eines Elements oder Feldes.
Cols	Anzahl der Schriftzeichen in einer Zeile innerhalb eines Textfeldes
Rows	Anzahl der Zeilen eines Textfeldes.
Size	Anzahl der Zeichen eines Textfeldes.

3.2.7 Verwendung des FormModels

Wie im obigen Beispiel gezeigt, müssen die einzelnen Felder eines Formulars registriert werden. Um ein Formular anzuzeigen sollte für den Formularkopf die Methode `renderFormOpen()` benutzt werden. Sie erzeugt neben einem validen HTML-Code auch die „Action“ an die das Formular gesendet wird. Damit jedoch Vimerito selbst weiß wohin die Formulardaten gesendet werden müssen, ist ein die Konfiguration des Formulars selbst notwendig. In der erstellten Formularklasse wird dafür das Array `Form` im Konstruktor angepasst. Folgende Einstellungen werden verlangt:

`'type'`	Gibt an, ob Daten oder Dateien verschickt werden sollen. Bis Version 0.2 von Vimerito wird jedoch nur die Option `'data'` unterstützt.
`'action'`	Diese Option verlangt ein Array. Darin angegeben wird der Controller und die Methode an die Daten geschickt werden sollen. Das Array entspricht folgender Form: `array( 'controller' => 'myController', 'method' => 'myMethod' );`
`'method'`	Gibt die Methode an, mit der die Daten versendet werden. Sprich ob als POST oder im URL als GET. Bis Vimerito 0.3 wird jedoch nur die Methode `'POST'` unterstützt.

Sind diese Einstellungen vorgenommen, kann das Formular ausgegeben werden. Eine Beispielhafte Ausgabe unseres Registrierungsformulars könnte dann wie folgt aussehen:

```
<?
$myRegister = new Register;
$myRegister->renderFormOpen();
?>
<table>
    <tr>
     <td><?=$myRegister->renderLabel("email");?></td>
          <td><?=$myRegister->renderField("email");?></td>
    </tr>
    <tr>
```

```
     <td><?=$myRegister->renderLabel("pw1");?></td>
     <td><?=$myRegister->renderField("pw1");?></td>
    </tr>
    <tr>
     <td><?=$myRegister->renderLabel("pw2");?></td>
     <td><?=$myRegister->renderField("pw2");?></td></tr>
    <tr>
          <td></td>
          <td><?=$myRegister->renderField("submited");?></td>
    </tr>
</table>
</Form>
```

Das FormModel kann außerdem dafür genutzt werden die vom Benutzer versandten Daten auf Gültigkeit zu prüfen. Dafür muss das Array `Criteria` im Konstruktor der Formularklasse angepasst werden. Die Felder, die überprüft werden sollen, werden in dem Array als Schlüssel angegeben. Dann wird für dieses Feld ein Kriterium nach dem es geprüft werden soll festgelegt. Die einfachste Form auf unser Beispiel bezogen wäre:

```
$this->Criteria = array(
    'email'         =>   'required'
);
```

Das sagt dem FormModel, dass das Feld 'email' ausgefüllt sein muss. Ein Kriterium kann aber auch aus einem Array bestehen, in dem zunächst ein Operand und dann ein Suchmuster angegeben wird. Als Beispiel:

```
$this->Criteria = array(
    'email'         =>   array('is', 'email')
);
```

Der Operand 'is' oder die Negierung 'isnot' werden nur im Zusammenhang mit den Prüfmustern verwendet. Folgende Prüfmuster stehen zur Auswahl:

'text'	Text, Zahlen und die Sonderzeichen ! ? “ ' . sind erlaubt.
'textonly'	Nur Text und alle Sonderzeichen sind erlaubt.
'number'	Nur Zahlen sind erlaubt.
'email'	Der Text muss eine gültige high-level Emailadresse sein.

Es ist aber auch möglich Formulardaten untereinander zu vergleichen. Das macht beispielsweise bei der Registrierung von Passwörtern sind. Die Validierung ist nur erfolgreich, wenn 'pw1' gleich 'pw2' ist.
Das lässt sich als Kriterium folgendermaßen ausdrücken:

```
$this->Criteria = array(
    'pw1'      =>   array('equal', 'pw2')
);
```

Als erstes wird in dem Kriterium der Vergleichsoperator angegeben und dann das Feld mit dem verglichen werden soll. Folgende Operanden stehen zur Auswahl:

`'equal'`	Es wird auf Gleichheit geprüft.
`'nequal'`	Es wird auf Ungleichheit geprüft.

'='	Entspricht equal.
'!='	Entspricht nequal.
'bigger'	Das zu prüfende Feld muss größer dem Vergleichsfeld sein.
'smaller'	Das zu prüfende Feld muss größer dem Vergleichsfeld sein.
'>'	Entspricht bigger.
'<'	Entspricht smaller.

Die letztendliche Validierung erfolgt in die Empfangsmethode, an welche die Daten gesendet werden.

```
class myController extends VController{
    public function myMethod(){
     myRegister = new Register;
     if(myRegister->validate() == true){
          echo 'Alle Daten sind korrekt!';
     }else{
          echo 'Sie haben eine fehlerhafte Eingabe gemacht!';
     }
    }
}
```

Für jedes Feld lassen sich natürlich auch mehrere Kriterien angeben. Nur wenn ein Feld allen Kriterien entspricht gilt es als valide.

3.3 Der Controller

Der Controller stellt bei jedem MVC die Programmlogik bereit. Außerdem bietet der Controller den Einstiegspunkt beim Ausführen der Anwendung bzw. der Webseite. Vom Controller wird entschieden welche Models oder Views (mehr zu Views findest du im Abschnitt „Der View“) geladen werden.
Jeder Controller in Vimerito leitet sich von der Klasse `VController` ab. Dabei muss jeder Controller einen Konstruktor und eine Einstiegsmethode enthalten. Ansonsten ist ein Controller konfigurationsfrei.
Ein Controller-Datei sollte folgende Vorgaben erfüllen:
Die Datei sollte im Ordner `DOCUMENT-ROOT/application/controller` liegen oder in einem Unterordner. Der Name einer solchen Controllerklasse sollte folgende Konventionen erfüllen:

1. Groß- und Kleinschreibung wird beachtet
2. Der Dateiname sollte folgendes Format haben: {Klassenname}{„Controller“}.Endung

   Ein Beispiel: MyController.class.php
3. Das Zeichen „_“ zeigt Vimerito an, dass es einen Unterordner gibt. So liegt die Klasse „my_class_file“ in der Datei `DOCUMENT-ROOT/application/controller/my/class/fileController.class.php`

3.3.1 Verwendung des Controllers

Ein Controller könnte wie folgendes Beispiel aussehen:

```
class My extends VController{
    public function __construct(){
```

```
     parent::__construct();
     }

     public function MyInit(){
      //Abarbeitung der Anwendung
     }
}
```

Dies ist ein übersichtliches Grundgerüst für einen Controller. Die Methode `MyControllerInit` ist die Standardmethode. Diese wird aufgerufen, wenn im URL nur der Controller und keine Methode angegeben ist. Mehr dazu findest du im Abschnitt „Der Router“. Die Standartmethode eines Controller ist also der Name der Klasse mit einem anschließenden `Init`.
Der Controller bietet aber auch die Möglichkeit Methoden umzuleiten. Dies macht deine Klassen wiederverwendbarer. Für diesen Zweck existiert die Methode `registerMethodAlias`. Mit dieser Methode können Methoden anderer Klassen über einen neudefinierten Namen aufgerufen werden. Diese neuen Klassen müssen Vimerito nicht bekannt sein, sofern sie die allgemeinen Namenskonventionen einhalten. Aufgerufen werden können sowohl statische sowie auch nicht-statische Methoden. Einzige Voraussetzung ist, dass sie als `public`, öffentlich also, deklariert wurden. Die Methode erwartet als Parameter den neuen Namen der Methode und ein Array oder ein String, welcher den Klassnamen oder ein Objekt auf eine Klasse und den Methodennamen bereit hält.
Zur Veranschaulichung, welchen praktischen Nutzen eines solches Verhalten von Vimerito bringt, soll ein kleines Beispiel dienen.

Zunächst denken wir uns, dass wir eine Klasse schreiben, die Datumsfunktionen bereit stellt. Diese Klasse liegt im Controllerordner. Sie muss aber nicht von der Klasse VController abgeleitet werden, da sie auf keine Methoden dieser Klasse zurück greift.

```
class Datum{
     public static function DateTimeNow(){
      return time();
     }

     public function DateFormatet(){
      return date('d.m.y ');
     }
}
```

Wir haben statische Methode `DateTimeNow` und die Methode `DateFormatet` angelegt.
Nun schreiben wir einen Controller der auf diese beiden Methoden zugreifen will.

```
class my extends VController{
     public function __construct(){
      parent::__construct();
      $this->registerMethodAlias('dateString',
                 array('Datum', 'DateFormatet');

      $obj = new DatumController;
      $this->registerMethodAlias('Now', array($obj, 'DateTimeNow');
     }

     public function myInit(){
```

```
     echo $this->Now().“<br />“; // gibt etwas wie 25478964 aus.
     echo $this->dateString(); //gibt etwas wie 13.05.2011 aus.
    }
}
```

Wer seinen neu registrierten Methoden Parameter übergeben will, kann dies in unbegrenzter Zahl tun, ohne weitere Einstellungen tätigen zu müssen. Es wird aber ein Fehler ausgeworfen, wenn die Parameterzahl der Quellmethode nicht mit der, der aufgerufenen Methode über einstimmt.

Diese Methode ist sehr nützlich um oft verwendete Methoden nicht immer wieder als Quelltext einbinden zu müssen, sondern einfach auf diese zu verweisen.

Wer befürchtet bei der Namensvergabe seiner Methode mit dem Namen der Standartmethode zu kollidieren oder aus praktischen Gründen einen anderen Namen wählen möchte, der hat die Möglichkeit als Standartmethode eine andere zu verwenden, als eben beschrieben. Im Konstruktor seines Controllers muss dafür folgende Zeile eingefügt werden:

```
$this->_methods['default'] = 'meineMethode';
```

Als Standartmethode wird nun von Vimerito

```
$this->meineMethode();
```

auf gerufen.

3.3.2 Actions

Controllermethoden können über den URL angesteuert werden. So steht in der Stadartkonfiguration im URL nach der Webadresse zuerst der Controller und dann die Methode die aufgerufen werden soll.
„www.my-site.com/mycontroller/mymethod.html“ ruft den Controller „mycontroller“ und die Methode „myMethodAction“ auf.
Falls jetzt Fragezeichen auftauchen, bezüglich „Action“, muss an dieser Stelle erklärt werden, was eine Action ist.
Nur Actions können direkt vom Endbenutzer aufgerufen werden. Eine Action wird durch den Methodennamen mit einem anschließenden „Action“ kenntlich gemacht.

```
class my extends VController{
    public function myMethodAction(){
     //code
    }
}
```

Dieses Prinzip gilt nicht für die Init-Methode. Diese Methode bleibt davon unberührt.

3.3.3 Routing

Das Routing ist im Grunde genommen die Interpretation des URL. Wie eben beschrieben wird im URL der Controller und die Action festgelegt. Weiterhin können Parameter von einer Seite auf eine andere übergeben werden. Für diese Zwecke existiert die Klasse `VRouter`. Sie enthält alle Informationen über einen URL. Sprich:

- Der Controller der aufgerufen werden soll
- Die Action

- Sämtliche im URL angegebenen Parameter und deren Werte

Standardmäßig werden einzelne Parameter durch ein „/“ von einander getrennt. Jeder Parameter enthält eine Bezeichnung und einen Wert. Abschließend ist ein „.html“ zu setzen.

`/Parameter1-Wert1/Parameter2-Wert2.html`

Dieser Teil-URL enthält zwei Parameter:

- „Parameter1“ mit „Wert1“ als Wert
- „Paramert2“ mit „Wert2“ als Wert

Einzelne Parameter können durch die statische Methode `VRouter::getParam('Bezeichnung')` abgerufen werden.

```
VRouter::getParam('Parameter');
```

gibt in unserem Fall die Zeichenkette `'Wert1'` zurück.
Jedoch lassen sich auch alle übergebenen Werte in Form eines Arrays mit dem Befehl:

```
VRouter::getParamArray();
```

ausgeben.
Um valide Vimerito-URLs zuerhalten kann die Methode `Vimerito::createUrl(Array[, Array])` genutzt werden.
Das erste Array enthält dabei den Namen des Controllers und die Action. Das zweite Parameter je einen Schlüssel mit einem Wert.

```
Vimerito::createUrl(
    array(
     'registration',
     'show'
    ),
    array(
     'user'    =>   'new',
     'context' =>   'fromLogin'
    )
);
```

Dieser URL wird den Controller „registration“ und die Action „show“ aufrufen. An diesen Controller wird der Parameter „user“ mit dem Wert „new“ und der Parameter „context“ mit dem Wert „fromLogin“ übergeben. Der URL sieht dann wie folgt aus:
`http://my-site.com/registration/show/user-new/context-fromLogin.html`

In manchen Fällen ist es nützlich, den Benutzer automatisch zu einer zuvor besuchten Seite oder einer anderen Seite hin umzuleiten. Für diesen Zweck stellt die Klasse Vimerito die statische Methode `Vimerito::redirect([Code[, Array[, Array]]]);` zur Verfügung. Wird diese Methode parameterlos aufgerufen, wird der Benutzer zur letzten besuchten Seite inklusive aller Parameter umgeleitet. Als Umleitungscode wird in diesem Fall 307, „Temporary Redirect“ verwand. Als nächster Wert können Parameter und Werte an die Seite zu der umgeleitet wird, übergeben werden. Als letzter Parameter kann eine Controller und eine Action übergeben werden.

3.4 Der View

Views sind Ausgabevorlagen, auch Templates genannt, die während der Abarbeitung der Anwendung mit Inhalt gefüllt werden. In Vimerito existieren 2 Arten von Views.

- Der allgemeine View

- Das Layout

Zwischen beiden View-Arten gibt es einen signifikanten Unterschied. Während der View mit Variableninhalt gefüllt oder mit anderen Views kombiniert werden kann, kann das Layout nur mit Views befüllt werden. Dieses Konzept soll Redundanzen in Views mindern bzw. komplett verhindern können. Die Anzahl der Views die pro Seite angezeigt werden ist unbegrenzt. Es kann jedoch immer nur ein Layout geladen werden.
Ein View, egal ob es sich um ein Layout oder eine Viewdatei handelt, beinhalten XML oder HTML-Code.
Aus einem View heraus kannst du auf die Variablen deines Controllers zugreifen. Vimerito verfolgt dabei das Konzept, dass jede Variable, welche im Controller initialisiert wurde, auch im View verfügbar ist. Variablen brauchen also, nicht wie anderen Templatesystemen angemeldet werden. Dennoch existiert die Möglichkeit Werte aus anderen Quellen, die keine Controller sind, dem View bekannt zu machen.

View-Dateien sollten im Ordner `DOCUMENT-ROOT/application/views` oder in einem Ordner darunter liegen. Der Dateiname es Views sollte folgenden Konventionen entsprechen:

1. Groß- und Kleinschreibung wird beachtet
2. Der Dateiname sollte folgendes Format haben: {Viewname}.{„php“}

   Ein Beispiel: meinView.php

Letztlich bleibt die Benennung deiner View-Dateien dir überlassen, da Vimerito der gesamte Dateiname übergeben werden muss.

3.4.1 Verwendung von Views

Um einen View nutzen zu können, muss eine Objekt der Klasse VView angelegt werden. Der Konstruktor kann parameterlos aufgerufen werden oder es wird der Dateipfad zum View übergeben.

```
$view = new VView(“pfad/zum/view.php”);
//oder
$view = new VView;
$view->load(“pfad/zum/view.php”);
```

Um Variablen in einem View nutzen zu können, muss diese Variable mit einem eindeutigen Namen dem View bekannt gemacht werden. Dies erfolgt über die Methode `assignVar`. Als Parameter muss ein Name, mit dem der Wert später im View aufgerufen werden kann, angegeben werden und den Wert selbst.

```
$view->assignVar('name', 'wert');
```

Es kann aber auch ein Array mit einer Sammlung von Namen und Werten an die Methode übergeben werden.

```
$view->assignVar(array(
    'name1'=>'wert1',
    'name2'=>'wert2'
));
```

Im View erfolgt der Zugriff auf diese Wert mittels der PHP-Kurzschreibweise:

```
<html>
    <head></head>
    <body>
     <div>
          <?=$this->name1;?>
```

```
     </div>
     <div>
          <?=$this->name2;?>
     </div>
    </body>
</html>
```

Um einen View auszugeben gibt den Befehl:

```
$view->render([cacheMode[, cachingtime]]);
```

Für das Rendern von Views gibt verschiedene Cachemodes. Diese werden optional übergeben.

`CacheToFile`	Die Ausgabe wird in einer Datei gespeichert und dann als HTML ausgegeben. Der 2. Parameter ist diesem Fall der Wert, der die Haltbarkeit des gespeicherten Ergebnisses in Sekunden angibt. Bis zum Ablauf der Haltbarkeit wird nicht mehr der View berechnet, sondern nur noch die Vorberechnete Datei geladen und ausgegeben.
`CacheToVar`	Die Ausgabe erfolgt in eine Variable. Diese Variable lässt sich später abrufen und es können nachträglich noch Änderungen an der Ausgabe vorgenommen werden. Ist der 2. Parameter `true`, wird das Ergebnis ausgegeben. Wenn es `false` ist (Standard) wird das Ergebnis nicht ausgegeben. Die gecachte Ressource lässt sich über das Attribut `cachedView` ansprechen.

Hier ein kleines Beispiel für die Verwendung eines Views.
In einem Controller wird der Variable `$text` ein Beispieltext übergeben. Dieser soll in einem View ausgegeben werden. Der Controller ist wie folgt aufgebaut:

```
<?
    class my extends VController{
     public $text = '';
     public function __construct(){
          parent::__construct();
     }

     public function myInit(){
          $text = '
               Dies ist ein Beispieltext. Dieser soll in einem
               View angezeigt werden. Das ist mit Vimerito 2
               total einfach!';
          $myView = new VView();
          $myView->load('myView.php');
          $myView->assignVar('text', $text);
          $myView->render();
     }
    }
?>
```

Der View `myView.php` könnte wie folgt aussehen:

```
<html>
```

```
    <head>
     <title>MyView</title>
    </head>
    <body>
     <?=$this->text;?>
    </body>
</html>
```

Die HTML-Ausgabe, die mit diesem Code erzeugt wird ist:

```
<html>
    <head>
     <title>MyView</title>
    </head>
    <body>
     Dies ist ein Beispieltext. Dieser soll in einem View
     angezeigt werden. Das ist mit Vimerito 2 total einfach!
    </body>
</html>
```

3.4.2 Verwendung des Layouts

Das Layout wird verwendet, um die Grundstruktur einer Webseite zu laden. Ein Layout ist ein HTML-Dokument welches im Ordner DOCUMENT-ROOT/application/layout liegt, beziehungsweise in einem Unterordner. Für dieses Layout können sog. Blöcke definiert werden. Bei Blöcken handelt es sich um HTML-Elemente, in die andere Viewdateien eingesetzt werden. In der Regel werden für diesen Zweck DIV-Container verwendet. Anhand des CSS-Selektors können diese Blöcke registriert und genutzt werden. CSS-Selektoren sind beispielsweise aus JavaScript bekannt. Die Syntax entspricht sogar der aus JavaScript und bekannten Frameworks wie jQuery. Wichtig ist, dass kein JavaScript genutzt wird, um das Layout zu befüllen. Es wird ausschließlich PHP genutzt. Die Steuerung der Blöcke bzw. des Layouts erfolgt im Controller.
Beispiellayout:

```
<html>
    <head>
     <title>Layout</title>
    </head>
    <body>
     <div id='content'>
     </div>
     <div id='menu'>
     </div>
    </body>
</html>
```

Das Layout wird als `index.html` im Layoutordner gespeichert.
In diesem Layout sind zwei Bereiche definiert. Ein DIV-Container mit der ID `content` und einer mit der ID `menu`.
`content` wird für Inhalt und `menu` für ein Seitenmenü verwendet. Im nächsten Schritt werden 2 Viewdateien vorbereitet. Die eine Viewdatei enthält den Inhalt, der auf der Seite dargestellt werden soll und eine Viewdatei enthält ein Seitenmenü in Listenform.
Für den Contentbereich erstellen wir die Viewdatei `content.php` im Viewordner.

```
<h1>Mein Inhalt</h1>
Willkommen auf dieser Seite. Ich möchte dir den Inhalt vorstellen.
```

Das Menü kommt in die Datei `menu.php` in den Viewordner.

```
<ul>
    <li>Link1</li>
    <li>Link2</li>
</ul>
```

Als letztes wird der Controller programmiert. In ihm wird das Layout mit den Viewdateien bekannt gemacht. Beispielhaft sähe der Controller wie folgt aus.

```
<?
    class my extends VController{
    public function __construct(){
         parent::__construct();
    }
    public function myInit(){
         $blocks = array(
              'content' =>   '#content',
              'menu'    =>   '#menu'
         );
         VLayout::load('index.html');
              $inhaltView = new VView('content.php');
               $inhaltView->render(CacheToVar);

              $linkView = new VView('menu.php');
               $linkView->render(CacheToVar);

              VLayout::insertIntoBlock('content',
                                  $inhaltView->cachedView);
              VLayout::insertIntoBlock('menu',
                                  $linkView->cachedView);

              VLayout::renderLayout();
    }
    }
?>
```

Zunächst werden die Viewdateien über ein Objekt der Klasse VView mit dem Befehl `load` geladen. Dann wird ein Array erstellt welches die Blöcke enthält. Es wird ein Name für den Block vergeben und der CSS-Selektor als Wert angegeben.
Im nächsten Schritt wird das Layout geladen. Der 1. Parameter ist der Name der Datei im Layoutordner und der 2. optionale Parameter sind die Blöcke, die registriert werden sollen.
Nach Registrierung können die Viewdateien in das Layout kopiert werden. Dafür müssen diese mit dem Parameter `CacheToVar` gerendert werden.
Mit dem Befehl insertIntoBlock werden die gecachten Viewressourcen in die entsprechenden Blöcke kopiert.
Das gesamte Layout kann nun gerendert werden. Der ausgegebene HTML-Code sieht nun wie folgt aus:

```
<html>
    <head>
     <title>Layout</title>
    </head>
    <body>
     <div id='content'>
          <h1>Mein Inhalt</h1>
          Willkommen auf dieser Seite. Ich möchte dir den Inhalt vorstellen.
     </div>
     <div id='menu'>
          <ul>
               <li>Link1</li>
               <li>Link2</li>
          </ul>
     </div>
    </body>
</html>
```

Das eben angeführte Beispiel ist doch recht lang, und zum Teil unübersichtlich. Außerdem enthält es viel sich wiederholenden Code. Was das Einfügen in das Layout angeht, gibt es eine kürze Schreibweise:

```
$view->sendToLayout('ElementId');
```

Mit diesem Befehl wirkt unser Beispiel gleich viel angenehmer für die Augen.

```
<?
    class my extends VController{
     public function __construct(){
          parent::__construct();
     }
     public function myInit(){
          $blocks = array(
               'content' =>   '#content',
               'menu'    =>   '#menu'
          );
          VLayout::load('index.html');
               $inhaltView = new VView('content.php');
                 $inhaltView->sendToLayout("content");

               $linkView = new VView('menu.php');
                 $linkView->sendToLayout("menu");


               VLayout::renderLayout();
     }
    }
?>
```

4. Arbeit mit Sessions

Für die Arbeit mit Sessions steht in Vimerito 2 die Klasse VSession zur Verfügung. Das besondere ist, dass alle Änderung in der Session nur temporär geschehen, bis der Controller komplett

abgearbeitet ist. Erst dann werden alle Änderung endgültig übernommen.

Vimerito 2 erstellt automatisch beim Starten des Controllers eine neue Session. Um Werte in die Session zu speichern gibt es den Befehl `VSession::set('name', 'wert')` und zum auslesen von Werten der Befehl `VSession::get('name')`. Soll eine Session gelöscht werden rufe den Befehl `VSession::destroy()` auf und wenn die Session-Id neu generiert werden soll den Befehl `Vsession::regenerateID()`. Mit dem Befehl `VSession::saveSession()` werden alle temporären Änderungen an der Session übernommen.

Bitte beachte, dass wenn du den Benutzer per Redirect zu einer anderen Seite umleitest alle Änderungen an der Session verloren gehen. Vor der Umleitung solltest du die Session speichern.

```
<?
…
VSession::set('benutzer', 'Franz');
VSession::save();
echo VSession::get('benutzer'); //gibt 'Franz' aus
VSession::regenrateID(true); //generiert eine neue SessionID –
alle
//Daten bleiben erhalten
echo VSession::get('benutzer'); //gibt Franz aus.
VSession::regerateID(); // generiert eine neue SessionID – alle
//Daten werden gelöscht.
echo VSession::get('benutzer'); // gibt eine leere Zeichenkette
aus.
VSession::destroy(); //löscht die aktuelle Session.
...
?>
```

5. Authentifizierung

Vimerito 2 beinhaltet seit Version 0.2 ein assoziatives Rechtesystem. Dieses lässt sich schnell anpassen und automatisieren. Es lassen sich unendlich viele Benutzerkonten mit unterschiedlichen Zugriffsrechten anlegen.

5.1 Anlegen von Benutzerkonten

Die Datei für die Konfiguration der Benutzerkonten liegt in `DOCUMENT-ROOT/application/configuration` und heißt „accessConfiguration.php“. In dieser Datei werden 4 Variablen initialisiert.

- $__cachedAccessKeys
- $__cachedAccessRights
- $__cachedNameforAccessKey
- $__cachedRedirectController

5.1.1 Einrichten der Benutzerkonten

Jedes Benutzerkonto benötigt einen eindeutigen Namen und wird im Array `$__cachedAccessKeys` eingetragen.

```
$__cachedAccessKeys = array(
    0     =>    'Guest',
    1     =>    'Registered',
    2     =>    'Administrator'
);
```

Das Konto mit den wenigsten Rechten sollte den Index 0 bekommen. Alle anderen Konten müssen keiner Ordnung entsprechen. Nach dem Eintragen sind die Konten dem System bekannt.
Im nächsten Schritt muss festgelegt werden, was ein Benutzer sehen darf und wie sich seine Zugriffsrechte definieren.
Da in Vimerito 2 ein assoziatives Rechtesystem integriert ist, definieren sich die Zugriffsrechte über die Rechte andere Konten. Als Beispiel:

1. Ein Gast darf nur Zugriff auf Seiten die für Gäste sind
2. Ein Benutzer hat Zugriff auf die Seiten die für Gäste und für Benutzer sind
3. Der Administrator hat Zugriff auf die Seiten, die für Gäste, Benutzer und den Administrator vorgesehen sind.

Diese Verkettung findet sich im Array `$__cachedAccessRights` wieder. Die Zugriffsrechte werde wie folgt vergeben:

```
$__cachedAccessRights = array(
    'Guest'          =>    array(),
    'Registered'     =>    array('Guest'),
    'Administrator'  =>    array('Guest', 'Registered')
);
```

Der Schlüssel in diesem Array entspricht also dem Benutzerkonto, dem Rechte zu gewiesen werden soll. Das folgende Array als Wert des Schlüssels, beinhaltet alle Konten, deren Rechte dieses Benutzerkonto einschließt.

Die Variable `$__cachedNameforAccessKey` gibt an, unter welchem Namen die Berechtigung des Benutzers in der Session abgelegt wird. Als letztes kann in dem Array `$__cachedRedirectController` ein Controller und eine Action hinterlegt werden, zu welcher der Benutzer im Fall fehlender Berechtigung umgeleitet wird. Dies ist nur relevant, wenn die Authentifizierung automatisch erfolgt. Die automatische Authentifizierung kann in der Datei `applicationConfiguration.php` angeschaltet werden.

5.2 Zugriffsberechtigung im Controller

Um für einen Controller bzw. für eine Action Zugriffsrechte ein zu richten, muss im Konstruktor des Controllers das Array `accessOption` angepasst werden. Jeder Schlüssel des Arrays entspricht der Action und der Wert entspricht der Mindestberechtigung, die ein Benutzer haben muss, um Zugriff auf diese Action haben zu können.
Haben wir einen Controller mit 2 Actions:

- show
- edit

wobei `show` für jeden Benutzer und jeden Gast zugänglich sein soll, jedoch nur Benutzer auf die Action `edit` zugreifen dürfen, passen wir `accessOption` wie folgt an:

```
...
public function __construct(){
    parent::__construct();
```

```
    $this->accessOption = array(
     //die Action show ist für Gäste und Benutzer zugänglich
     'showAction'   =>   'Guest',
     // die Action edit ist nur für registrierte Benutzer
     //zugänglich
     'editAction'   =>   'Registered'
    );
}
...
```

Für die Initialmethode (Init) gilt das diese ebenfalls voll ausgeschrieben werden muss:

```
...
public function __construct(){
    parent::__construct();
    $this->accessOption = array(
     'controllerNameInit'     =>   'Guest'
    );
}
…
```

Ist die automatische Authentifizierung ausgeschaltet, kann auch manuell geprüft werden. Mit `VaccessRights::authenticateUser('methodname');` kann dies geprüft werden. Diese Methode gibt `true` zurück, wenn der Benutzer über die entsprechende Berechtigung verfügt und `false`, wenn die Authentifizierung fehl schlägt.

6. Weiterführende Arbeit mit Views

Vimerito 2 erlaubt ein dynamisches Arbeiten mit Views. So können Views während der Laufzeit verändert und angepasst werden. Das gilt auch für das Layout.

Die Klasse VHtmlElement ist eine objektorientierte Form eines normalen Html-Elementes. Mit dieser Klasse können neue Html-Elemente erstellt, mit Inhalt gefüllt und später in einen View eingefügt werden. Dies eröffnet vielfältige Möglichkeit bei der Programmierung.

```
…
$element = new VHtmlElement();
$element->tag = "div";
$element->innerText = "Ein dynamischer Text!";
$element->insert($ViewObjekt, Append, "#ElementId");
…
```

Dieser Quelltext erstellt einen Div-Container, der den Text „Ein dynamischer Text!“ enthält. Dieses Element wird in einen View eingefügt. In dem View wird nach einem Element mit der Id „ElementId“ gesucht und dort hinein kopiert.

Der View könnte im Ergebnis wie folgt aussehen:

```
…
<body>
<p>Fließtext ohne Sinn!</p>
<div id="ElementId">
    <div>Ein dynamischer Text!</div>
```

```
</div>
</body>
…
```

Es ist egal ob der View schon berechnet wurde oder nicht. Nur wenn der schon in ein Layout eingefügt wurde, wird im Ergebnis nichts zu sehen sein.

Soll das Element jedoch in das Layout eingefügt werden, muss der `Insert`-Befehl, wie folgt umgeschrieben werden:

```
…
$element->insert(Layout, Append, "#ElementId");
…
```

In diesem Fall wird die Konstante `Layout` genutzt.
Folgende Attribute können bei einem Html-Element gesetzt werden:

<table>
<tr><td>id</td><td>Legt die Id eines Elementes fest.</td></tr>
<tr><td>name</td><td>Legt den Namen eines Elementes fest.</td></tr>
<tr><td>parent</td><td>Legt mit einem CSS-Selektor fest, welchem HTML-Element das aktuelle Element untergeordnet ist.
Beispiele:<table>
<tr><td>body</td><td>Das Element wird in das Body-Element eingefügt.</td></tr>
<tr><td>.cssKlasse</td><td>Das Element wird in ein Element eingefügt, dass die CSS-Klasse „cssKlasse“ hat.</td></tr>
<tr><td>div#meinElement</td><td>Das Element wird in einen Div-Container eingefügt, dass die Id „meinElement“ besitzt.</td></tr>
</table></td></tr>
<tr><td>innerText</td><td>Legt fest welchen Text das Element umschließt. Natürlich kann auch HTML-Code verwendet werden.</td></tr>
<tr><td>class</td><td>Legt fest, nach welcher CSS-Klasse das Element formatiert werden soll.</td></tr>
<tr><td>style</td><td>In diesem Attribute können weitere CSS-Eigenschaften festgelegt werden.</td></tr>
<tr><td>src</td><td>Legt beim IMG-Tag fest, welche Source verwendet werden soll. Relative Pfade gehen vom Base-Verzeichnis aus.</td></tr>
<tr><td>attributes</td><td>Ist ein Array, auf dass es keinen direkten Zugriff gibt. Mit der Methode <code>addAttribute()</code> können weitere Attribute, wie <code>width</code> oder <code>height</code> hinzu gefügt werden, oder mit <code>removeAttribute()</code> entfernt werden.</td></tr>
</table>

Die Methode addAttribute fügt weitere Attribute hinzu, die im heutigen Standard eher mit CSS konfiguriert werden.

```
…
$element->addAttribute('width', '150');
…
```

Auf diese Weise wird das Attribut width auf 150 Pixel gesetzt. Jedoch wird mit

```
…
$element->removeAttribute('width');
…
```

das Attribut wieder entfernt.

7. Arbeiten mit JavaScript

Vimerito 2 versucht die Arbeit mit JavaScript auf ein Maximum zu vereinfachen. Sowohl das Einbinden eigener JavaScript-Bibliotheken, als auch das Einbinden eigenen Codes ist mit minimalen Aufwand verbunden.
Um JavaScripte in den eigenen Code ein zu binden, wird die Klasse VJavaScript benötigt. Diese Klasse arbeitet ähnlich wie Views. Es gibt mehrere Arten diese Klasse zu verwenden.

```
// Direkter Aufruf ohne Parameter. Anschließend wird ein Code
// eingebunden.
…
$js = new VJavaScript();
$js->setCode(
    "//JavaScript-Code"
);
// Aufruf mit Parameter. Ein Code wird gleich im Konstruktor fest
// gelegt.
…
$js = new VJavaScript('script://JavaScript-Code');
…
// Aufruf mit Parameter. Es wird eine Script-Datei übergeben,
// sich im JavaScript-Ordner „js“ in der aktuellen Applikation
befindet.
…
$js = new VJavaScript('file:pfad/zur/datei.js');
…
```

JavaScripte müssen wie Views auch anschließend noch an das Layout gesendet werden.

```
$js->sendToLayout();
```

Bei JavaScripten gibt es eine weitere Gemeinsamkeit zu den Views. Es können innerhalb von JavaScripten PHP-Variablen genutzt werden.

```
$js = new VJavaScript('file:pfad/zu/script.js');
$text = 'Dies ist ein PHP-Text';
$js->assignVar('phpText', $text);
$js->sendToLayout();
```

Die Variable phpText lässt ebenso wie in Views verwenden, wie dieser kurze JavaScript-Schnipsel zeigt.

```
$(document).ready(function(){
    $('element').text('<?=$this->phpText;?>');
});
```

8. Ressourcen

Vimerito 2 behandelt Views, Layouts und HTML-Elemente, die in Vimerito 2 erstellt werden als Ressourcen. Für diesem Zweck existiert eine abstrakte Klasse `VRessource`. Sie bildet eine Art Schnittstelle.

Diese Klasse ist entsprechend der Abstraktion keine funktionsfähige Klasse, sondern legt nur fest welche Methoden in einer abgeleiteten Klasse vorhanden sein müssen. Diese Methoden müssen in den ableitenden Klassen implementiert werden. Dadurch entstehen einheitliche Methodenbezeichnungen, die ein flexibles Erweitern von Ressourcenklassen möglich machen.

In Version 0.6.1 existieren in Vimerito 3 Arten von Ressourcen:

- VviewRessource
- VCachedViewRessource
- VLayoutRessource
- VHtmlRessource
- VJavaScriptRessource
- VCachedJavaScriptRessource

9. Arbeiten mit Applications

In Vimerito 2 ist es möglich eine Internetseite in verschiedene Applikationen zu unterteilen. So ist es beispielsweise möglich Frontend und Backend einer Internetseite von einander zu trennen. Jede Applikation lässt sich über den URL ansteuern. Eine Application sollte in einem Unterordner der `DOCUMENT-ROOT` liegen.

Applikationen müssen untereinander bekannt gemacht werden. Dies erfolgt in der Datei `DOCUMENT-ROOT/[application]/configuration/applicationConfiguration.php`.

Es wird das Array `applications` erzeugt. In diesem Array müssen jeweils alle anderen Applikationen eingetragen werden, nicht aber die Applikation selbst. In der Frontend-Applikation wird die Backend- bekannt gemacht und anders herum.

`applications` ist ein Array dessen Schlüssel dem Namen der Applikation entspricht und dessen Wert dem Pfad, ausgehend von der `DOCUMENT-ROOT`.

Eine Beispielkonfiguration könnte wie folgt aussehen:

```
$__cachedApplicationConfiguration = array(
    …
    'applications'   =>array(
                    'backend'  =>   'backenddir',
                    'onemore'  =>   'onemoredir'
                ),
    ...
);
```

Es können nur Methoden und Actions innerhalb einer Applikation auf gerufen werden.

Existieren 2 Applikationen „frontend“ und „backend“ und haben beide Applikationen eine Klasse „userController“, so wird „frontend“ nur auf seinen eigenen „userController“ zugreifen können. So kommt es zu keinen Kollisionen zwischen den verschiedenen Applikationen.

Wird die Methode Vimerito::createUrl verwendet, bezieht sich diese Methode automatisch auf die aktuelle Applikation.

10. Module

Module sind eigenständige Programmteile innerhalb einer Applikation. Module befinden sich generell in dem Ordner `DOCUMENT-ROOT/{Applikation}/modules`. Jedes Modul muss sich in einem eigenen Unterordner befinden. Ein Modul besitzt eine Hauptklasse, deren Name dem Namen des Moduls entspricht. Auch der Name des Unterordners in dem sich das Modul befindet muss dem Namen der Klasse gleichen. Wenn wir also eine Modulklasse anlegen, die wie folgt auf gebaut ist:

```
<?php
    class hello extends VModule{
          public function __construct(){
               parent::__construct();
          }
    }
?>
```

muss sich diese Datei mit dem Namen `helloModul.class.php` in dem Ordner `DOCUMENT-ROOT/{Applikation}/modules/hello` befinden. Das oben gezeigte Beispiel ist im Übrigen alles, was in dieser Klasse enthalten ist. Mehr muss nicht programmiert werden. Jedes Modul kann eine unendliche Zahl an Controllern haben, die sich wie aus bis jetzigen Programmierung mit Vimerito 2 bekannt in dem Ordner controllers innerhalb des Modul-Ordners befinden müssen, also `DOCUMENT-ROOT/{Applikation}/modules/hello/controllers`. Genau so verhält es sich mit Views. Ein Modul kann auch ein eigenes Layout besitzen.

Zu beachten bei der Programmierung mit Modulen ist, das Controller des Moduls zwar auch Klassen außerhalb des Moduls zugreifen können, etwa Helfer-Klassen (siehe nächstes Kapitel), andersherum ist dies aber nicht möglich. Ein Controller einer Applikation kann nicht auf Klassen eines Moduls zugreifen.

Über den Browser angesteuert wird ein Modul wie folgt:

http://www.seitenname.de/[applikation]/module/nameDesModuls/controller/methode/parameter1-wert1.html

In unserem Beispiel hieße das:

http://www.seitenname.de/module/hello/controller/methode/parameter1-wert1.html

11. Helper-Klassen

Helper-Klassen sind ein Konstrukt, die helfen sollen den Programmcode besser zu strukturieren. Es soll vermieden werden übermäßigen Ballast in Controller-Klassen auf zu nehmen. In Helper-

Klassen findet all das Platz, dass nicht direkt von Außen, vom Benutzer einer Webseite ansteuerbar sein soll. Außerdem lassen sich Methoden definieren, die nicht ständig im Gebrauch sind und durch diese Trennung nicht immer mit geladen werden müssen.

Helper-Klassen befinden sich in dem Ordner `DOCUMENT-ROOT/[application]/helpers/` und müssen im Klassennamen auf „Helper“ enden. Der Dateiname entspricht dem Klassennamen inklusive dem „Helper“-Suffix.

Die Klasse `logHelper` befindet sich also in der Datei `logHelper.class.php`.

12. Sprachunterstützung

Ab Vimerito 0.6 ist es möglich unterschiedliche Sprachen zu integrieren. Dies wird durch die System-Helper-Klasse `VLang` ermöglicht. Es ist einerseits möglich mit INI-Dateien zu arbeiten oder mit Datenbanktabellen. In der Datei `DOCUMENT-ROOT/[application]/configuration/applicationConfiguration.php` wird unter dem Eintrag `defaultLanguage` ein frei gewähltes Kürzel für die Sprache hinterlegt, bspw. `'de'` oder `'eng'`. Des weiteren wird eine Standartquelle über den Schlüssel `defaultLanguageSource` definiert. Wird als Quelle eine INI-Datei verwendet wird dem Namen der Datei ein `file:` voran gestellt. Darauf folgt der Name der Datei, die sich dann im Ordner `DOCUMENT-ROOT/[application]/language/` befindet. Dem Dateinamen wird dabei je nach Sprache das Sprachkürzel voran gestellt. Hat man als Sprachkürzel Beispielsweise für Deutsch `de` definiert und gibt man als Dateinamen `lang` an, wird die Datei `DOCUMENT-ROOT/[application]/language/delang.ini` geladen.
Innerhalb der INI-Datei werden Schlüssel uns Werte definiert.

```
Welcometext = 'Hallo auf der Seite'
Back = 'Zurück'
```

In einer Datenbanktabelle müssen die Spalten `id`, `name`, `lang`, `string` definiert werden. Die Spalte `name` enthält den eindeutigen Namen einer Phrase, `lang` das Sprachkürzel und `string` die Phrase in der jeweiligen Sprache. `id` ist eine fortlaufende, eindeutige Zahl.
Um als Standartquelle eine Datenbanktabelle verwenden zu können, muss bei `defaultLanguageSource object:` und der Name eines Models angegebene werden. Die Konfiguration könnte dann wie folgt aussehen:

```
$__cachedApplicationConfiguration = array(
    ...
    'defaultLanguage'             => 'de',
     'defaultLanguageSource'      => 'object:languageModel'
);
```

Innerhalb eines Views kann dann die Methode `VLang::output` aufgerufen werden. Sie ermittelt automatisch die aktuelle Sprache oder lädt die Standartsprache. Dieser Methode wird lediglich der Name der Phrase übergeben. Über den Befehl `VLang::setLanguage` wird mit Übergabe des Sprachekürzels eine neue Sprache gewählt. Da dies in die Session des Benutzers geschrieben wird, brach dies nur einmal geschehen.
Mit dem Befehl `VLang::setSource` kann eine neue Quelle definiert werden. Handelt es sich bei dem übergebenen Wert um einen String, wird eine Dateiquelle angenommen. Handelt es sich

um die Instanz eines Models wird eine Datenbanktabelle angenommen. Wird dieser Befehl parameterlos auf gerufen, wird die Standartquelle geladen.

13. Funktionsreferenz

13.1 Vimerito.class.php

static Vimerito::__isSystemController		
Gibt anhand des Klassennamen an, ob die Klasse zu einem Controller vom System gehört, oder nicht.		
Sichtbarkeit:	public	
Parameter:	1. string Klassenname	Name der Controllerklasse
Rückgabe:	bool	Wenn es sich um einen Systemcontroller handelt wird true, ansonsten false zurück gegeben.

static Vimerito::addAliasClass		
Diese Funktion macht es möglich eine Klasse unter einem anderen Namen, als den eigentlichen Klassennamen an zu sprechen, unter einem Alias also.		
Sichtbarkeit:	public	
Parameter:	1. Array alias	Ist ein assoziatives Array. Folgende Werte müssen in diesem Array angegeben sein: 1. 'classname' – Name der Klasse, die aufgerufen werden soll 2. 'alias' – Neuer Name der Klasse. Dieser Name wird benutze um die Klasse auf zu rufen. 3. 'path' – Gibt den Pfad zu der Datei, die die Klasse enthält. Ausgegangen wird dabei vom Ordner: `DOCUMENT-ROOT`
Rückgabe:	bool	Im Erfolgsfall true. Wenn nicht alle geforderten Werte in dem Array angegeben wurden false.

static Vimerito::addUserClass		
Diese Funktion macht eine Klasse unabhängig von Namenskonventionen. Dabei wird der Name einer Klasse und der Pfad zu der Klassendatei ausgehend vom Basisverzeichnis angegeben. Beim Autoloading werden die Angaben dann berücksichtigt.		
Sichtbarkeit:	public	
Parameter:	1. string Klassenname 2. string Pfad	Name der Klasse Pfad zur Klassendatei ausgehend vom Basisverzeichnis.
Rückgabe:	bool	Im Erfolgsfall true. Wenn der Klassenname bereits hinzugefügt wurde, dann false.

static Vimerito::arrayToObject

Diese Methode wandelt ein Array in Objekt um und gibt dieses zurück.		
Sichtbarkeit:	public	
Parameter:	1. array	Ein Array, dass in ein Objekt umgewandelt werden soll.
Rückgabe:	object	Das umgewandelte Array in Objektform.

static Vimerito::createCode		
Erzeugt eine zufällige Zeichenkette und gibt diese zurück.		
Sichtbarkeit:	public	
Parameter:	1. integer	Anzahl der Zeichen, die die Zeichenkette haben.
Rückgabe:	string	Eine zufällige Zeichenkette.

static Vimerito::createUrl		
Erstellen einen Vimerito 2 konformen Url zu einem Controller und einer Action. Die Applikation wird dabei beachtet.		
Sichtbarkeit:	public	
Parameter:	1. Array, String	Sind in dem Array 2 Werte vorhanden, entspricht der erste Wert dem Controller und der zweite der Action. Befindet man sich in einer bestimmten Applikation wird diese ergänzt. Sind in dem Array 3 Werte vorhanden, entspricht der erste Wert der neuen Applikation in die gewechselt werden soll, der zweite dem Controller und der dritte der Action. Wird eine String übergeben wird diese wie sie ist in den Url eingesetzt.
	2. Array (optional)	Werte die an die auf zurufende Action übergeben werden sollen, wobei der Schlüssel dem Namen und der Wert dem Wert entspricht.
Rückgabe:	string	Eine Url inklusive „http://“

static Vimerito::getApplicationBuildingTime		
Gibt die Dauer in Sekunden an, die benötigt wurde, um die aktuelle Applikation abzuarbeiten.		
Sichtbarkeit:	public	
Parameter:	- keine -	
Rückgabe:	string	Dauer in Sekunden.

static Vimerito::returnApplicationArray	
Gibt ein Array mit der Konfiguration von Applications wie in der Datei `DOCUMENT-ROOT/{application}/configuration/applicationConfiguration.php` festgelegt, wobei {application} der Name der aktuellen Applikation ist.	
Sichtbarkeit:	public
Parameter:	- keine -

Rückgabe:	array	Entspricht dem Array `$__cachedApplicationConfiguration` das in der Konfigurationsdatei einer Applikation definiert wird.

static Vimerito::getApplicationName		
Gibt den Namen der aktuellen Applikation zurück.		
Sichtbarkeit:	public	
Parameter:	- keine -	
Rückgabe:	string	Name der aktuellen Applikation.

static Vimerito::getApplicationPath		
Gibt den Pfad der aktuellen Applikation zurück.		
Sichtbarkeit:	public	
Parameter:	- keine -	
Rückgabe:	string	Pfad zur aktuellen Applikation.

static Vimerito::getControllerPath		
Gibt den absoluten Pfad zum aktuellen Controller zurück.		
Sichtbarkeit:	public	
Parameter:	- keine -	
Rückgabe:	string	Pfad zur Controllerklasse.

static Vimerito::getInstance		
Gibt die Instanz eines bestimmten Objektes zurück. Folgende Parameter werden unterstützt: • CurrentApplication		
Sichtbarkeit:	public	
Parameter:	1. constant	Im Moment wird nur die Konstante `CurrentApplication` unterstützt.
Rückgabe:	object Instanz des geforderten Objektes	

static Vimerito::getVersion		
Gibt die aktuelle Versionsnummer von Vimerito 2 aus.		
Sichtbarkeit:	public	
Parameter:	- keine -	
Rückgabe:	string	Versionsnummer

static Vimerito::initApplication	
Initialisiert die Applikation. Dabei werden alle Konfigurationsdateien geladen, die Applikation, der Controller und die Action ausgelesen und die Session gestartet.	
Sichtbarkeit:	public
Parameter:	- keine -
Rückgabe:	- keine -

static Vimerito::loadApplicationConfiguration	
Lädt die Konfigurationsdatei ApplicationConfiguration.php der aktuellen Applikation.	
Sichtbarkeit:	public
Parameter:	- keine -
Rückgabe:	- keine -

static Vimerito::redirect		
Führt einen Redirect durch. Zu beachten ist dabei, dass zum Zeitpunkt des Aufrufes dieser Methode noch keine Ausgabe produziert worden sein darf.		
Sichtbarkeit:	public	
Parameter:	1. array	
Rückgabe:	string	Einen Url inklusive „http://“

static Vimerito::registerAutoloader		
Registriert einen weiteren Autoloader, wobei der Systemautoloader nach wie vor, die oberste Priorität hat.		
Sichtbarkeit:	public	
Parameter:	1. array	Callback des Autoloaders.
Rückgabe:	- keine -	

static Vimerito::runApplication	
Startet die aktuelle Applikation, das heißt, der geforderte Controller wird geladen und aufgerufene Action ausgeführt.	
Sichtbarkeit:	public
Parameter:	- keine -
Rückgabe:	- keine -

static Vimerito::setApplicationPath		
Setzt den Pfad zu einer Applikation. Alle Controller-Aufrufe werden ab diesem Zeitpunkt in diesem Ordner ausgeführt. Applikationen müssen in jeder anderen Applikation bekannt gemacht werden. Dies erfolgt in der Konfigurationsdatei: `DOCUMENT-ROOT/{application}/configuration/applicationConfiguration.php` festgelegt, wobei {application} der Name der aktuellen Applikation ist, im Array 'application'. In diesem Array wird der Schlüssel als Name der bekanntzumachenden Applikation angegeben und als Wert der Pfad zur Applikation. Dabei wird vom Wurzelverzeichnis `DOCUMENT-ROOT` ausgegangen.		
Sichtbarkeit:	public	
Parameter:	1. String applicationName	Der Name der Applikation.
Rückgabe:	- keine -	

static Vimerito::setJavaScriptMode	
Setzt den JavaScript-Modus. Wird dieser auf true gesetzt, wird automatisch das JavaScript-Framework „jQuery“ bei jeder Seite mit geladen.	
Sichtbarkeit:	public

Parameter:	1. bool Modus	true oder false
Rückgabe:	- keine -	

13.2 Controller.class.php

static VController::__call		
Dies ist die magische Methode __call. Sie wird stets aufgerufen, wenn eine Methode der Klasse aufgerufen wird.		
Sichtbarkeit:	public	
Parameter:	1. string Klassenname	Name der Controllerklasse
Rückgabe:	bool	Wenn es sich um einen Systemcontroller handelt wird true, ansonsten false zurück gegeben.

static VController::__construct	
Die magische Methode Konstruktormethode.	
Sichtbarkeit:	public
Parameter:	- keine -
Rückgabe:	- keine -

VController::addEventAfter		
Registriert für eine Methode der aktuellen Controller-Klasse eine Callback-Funktion, die nach deren Aufruf ausgeführt wird.		
Sichtbarkeit:	public	
Parameter:	1. string method	Name der Methode nach deren Aufruf die Callback-Funktion aus geführt werden soll.
	2. Array callback	Ein Array mit der Callback-Funktion. Enthält den Namen der Klasse und den Methodennamen.
Rückgabe:	- keine -	

VController::addEventBefore		
Registriert für eine Methode der aktuellen Controller-Klasse eine Callback-Funktion, die vor deren Aufruf ausgeführt wird.		
Sichtbarkeit:	public	
Parameter:	1. string method	Name der Methode vor deren Aufruf die Callback-Funktion aus geführt werden soll.
	2. Array callback	Ein Array mit der Callback-Funktion. Enthält den Namen der Klasse und den Methodennamen.
Rückgabe:	- keine -	

VController::getApplicationPath	
Synonym für Vimerito::getApplicationPath.	
Sichtbarkeit:	public

Parameter:	- keine -
Rückgabe:	String — Ist der Pfad zur aktuellen Applikation.

VController::registerMethodAlias		
Unter einem fiktiven Namen kann hier eine Methode einer anderen Klasse der aktuellen Klasse zu geordnet werden. Beim Aufruf wird die neue Methode behandelt, als sei sie Teil der aktuellen Klasse.		
Sichtbarkeit:	public	
Parameter:	1. String alias	Neuer Methodenname unter dem die Methode angesprochen werden soll.
	2. Array callback	Ein Array, dass den Klassennamen und den Namen der Methode enthält, die angesprochen werden soll.
Rückgabe:	Wirft einen Fehler aus, wenn `alias` bereits als Methode in der aktuellen Klasse existiert. Ansonsten keine Rückgabe.	

VController::run		
Startet die geforderte Methode. Ist die automatischen Authentifizierung aktiviert, wird noch geprüft, ob der Benutzer das Zugriffsrecht auf diese Methode hat.		
Sichtbarkeit:	public	
Parameter:	1. String method	Name der Methode, die ausgeführt werden soll.
Rückgabe:	- keine -	

13.3 VRouter

VRouter::__construct	
Konstruktor. Speichert zusätzlich den Referer in die Variable self::$__referer.	
Sichtbarkeit:	public
Parameter:	- keine -
Rückgabe:	- keine -

Static VRouter::route	
Dieser Befehl löst den URL in verschiedene Parameter auf. Er extrahiert die Applikation, den Controller, die Action und liest die zu übergebenen Parameter aus.	
Sichtbarkeit:	public
Parameter:	- keine -
Rückgabe:	- keine -

Static VRouter::calledController	
Gibt den Namen des aufgerufenen Controller zurück.	
Sichtbarkeit:	public
Parameter:	- keine -
Rückgabe:	String — Name des Controller.

Static VRouter::calledMethod		
Gibt den Namen des aufgerufenen Action (Method) zurück.		
Sichtbarkeit:	public	
Parameter:	- keine -	
Rückgabe:	String	Name der Action (Method).

Static VRouter::getParam			
Gibt einen Parameter zurück, der mit dem URL an die Seite übergeben wurde. Parameter sind klassisch an der Form `/parameter-wert/...html` zu erkennen.			
Sichtbarkeit:	public		
Parameter:	1.	String name	Name des Parameters.
Rückgabe:	Mixed		Wert des Parameters.

Static VRouter::getParamArray		
Gibt alle Parameter zurück, die an die Seite im URL übergeben wurden.		
Sichtbarkeit:	public	
Parameter:	- keine -	

Rückgabe:	Array	Parameter die an die Seite übergeben wurden inklusive deren Werte.
		Bsp.:
		`array(`
		`  'parameter1' => 'wert1',`
		`  'parameter2' => 'wert2'`
		`);`

13.4 VActivRecorder

VActivRecorder::analyseDatabase		
Analysiert die Tabellenstruktur einer Tabellen in einer Datenbank.		
Sichtbarkeit:	public	
Parameter:	Bool	Ist dieser Wert gleich TRUE wird die Tabellenstruktur gespeichert. Dieser Vorgang wird dann nur einmal ausgeführt. Ist dieser Wert FALSE wird die Tabellenstruktur bei jedem Aufruf erneut geprüft.
Rückgabe:	- keine-	

VActivRecorder::deleteWhere		
Löscht Daten aus der aktuellen Tabelle anhand einer Where-Klausel.		
Sichtbarkeit:	public	
Parameter:	Array	Die Where-Klausel als Array.

		Beispiel: array(array('id', '5', '=')) Dieses Beispiel würde Reihe löschen deren ID = 5 ist.
Rückgabe:	- keine-	

VActivRecorder::findAll		
Liefert alle Einträge in der aktuellen Tabelle zurück.		
Sichtbarkeit:	public	
Parameter:	Array	Optional. Ein Array, welches die Namen aller Spalten enthält die zurückgeliefert werden sollen. Beispiel: array('id', 'name', 'vorname')

	Array / String	Optional.
		Ein Array oder ein String, welcher die Order-Klausel enthält.
		Beispiel:
		array(
		  'name' => 'desc',
		  'vorname' => 'asc'
		)
		
		oder
		
		'ORDER BY `name` DESC, `vorname` ASC'

Rückgabe:	- keine-	

VActivRecorder::first	
Springt auf das erste Ergebnis einer Ergebnismenge.	
Sichtbarkeit:	public
Parameter:	- keine -
Rückgabe:	- keine-

VActivRecorder::send		
Schickt einen kompletten SQL-Query zur Datenbank. Achtung! Dieser Query wird nur zur Datenbank geschickt. Das heißt alle an zusprechenden Tabellen müssen in dem Query enthalten sein.		
Sichtbarkeit:	public	
Parameter:	String	SQL-Query

Rückgabe:	- keine-