Vimerito 2 MVC

Für Version 0.3 Alpha

Version des Handbuches 0.2

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.

Aktuell unterstützt Vimerito nur Mysql-Datenbanken.

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 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

3.1.1 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.

3.1.2 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.

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

3.1.3 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.

3.1.4 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.

3.2 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`

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.3 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.

3.3.1 Verwendung von Views

Views werden aus dem Controller heraus geladen. Für jeden View muss ein interner Name vergeben werden, mit welchem der View angesprochen werden kann.

```
$this->loadView('viewname', 'viewdatei.php');
```

Um einen oder mehrere Views auszugeben gibt es zwei Möglichkeiten. Zunächst lassen sich alle Views auf einmal via

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

oder jeder View einzeln mit dem Befehl:

```
$this->render('viewname'[, cacheMode[, cachingtime]]);
```

berechnen. Wird ein View einzeln gerendert gibt es einige Ausgabeoptionen (caching-modes). Diese werden optional als zweiter und dritter Parameter übergeben.

`CacheToFile`	Die Ausgabe wird in einer Datei gespeichert und dann als HTML ausgegeben. Der 3. 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 3. Parameter `true`, wird das Ergebnis ausgegeben. Wenn es `false` ist (Standard) wird das Ergebnis nicht ausgegeben.

Das Cachen ist mit dem Befehl `renderAll` nicht möglich. Hier ein kleines Beispiel für die Verwendung eines Views.
In einem Controller wird dem Attribut 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(){
               $this->text = '
                    Dies ist ein Beispieltext. Dieser soll in einem
                    View angezeigt werden. Das ist mit Vimerito 2
                    total einfach!';
```

```
            $this->loadView('myView', 'myView.php');
            $this->render('myView');
        }
    }
?>
```

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>
```

3.3.2 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(){
               $this->loadView('inhalt', 'content.php');
               $this->loadView('link', 'menu.php');

               $blocks = array(
                    'content'  =>    '#content',
                    'menu'     =>    '#menu'
               );
               VLayout::load('index.html');
               $this->render('inhalt', CacheToVar);
               $this->render('link', CacheToVar);

               Vlayout::insertIntoBlock('content', inhalt');
               Vlayout::insertIntoBlock('menu', 'link');

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

Zunächst werden die Viewdateien über den Befehl `loadView` 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 Views 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>
```

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.

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.