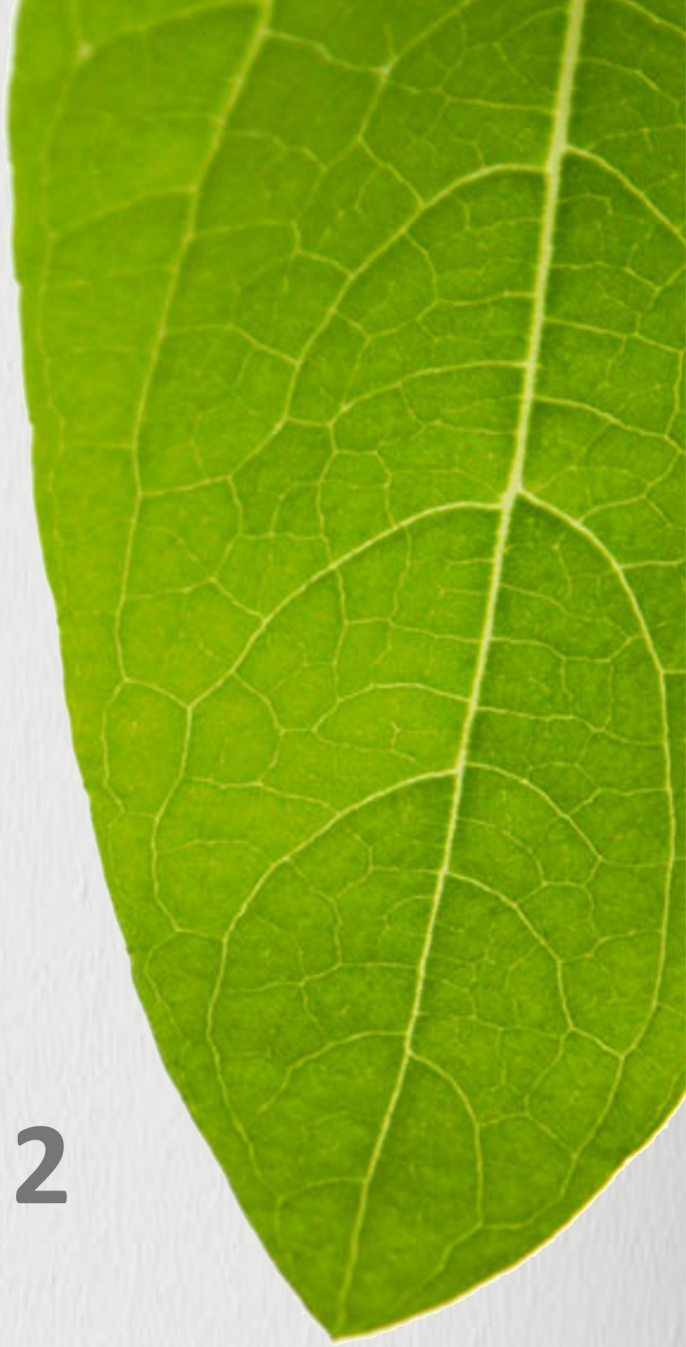
Zend Framework
Version 2.2



Webentwicklung mit Zend Framework 2

Grundlagen, Konzepte und praktische Anwendung

Michael Romer

Webentwicklung mit Zend Framework 2

Deutsche Ausgabe

Michael Romer

Dieses Buch können Sie hier kaufen http://leanpub.com/zendframework2

Diese Version wurde auf 2013-08-15 veröffentlicht

Inhaltsverzeichnis

1 Hallo, Zend Framework 2!

Hinfort mit all der grauen Theorie - schauen wir uns das Framework in Aktion an!

Wie im letzten Kapital besprochen, handelt es sich bei `Zend\Mvc` um eine eigenständige Komponente des Frameworks. Man kann es also nutzen, muss aber nicht. Eine "waschechte Zend Framework 2 Anwendung" setzt `Zend\Mvc` ein und dadurch wird der Anwendung auch die Verzeichnis- und Codestruktur in gewisser Form vom Framework diktiert. Das ist aber eigentlich ganz praktisch: Kennt man erst mal eine Zend Framework 2 Anwendung, kann man sich sehr schnell auch in anderen, ebenfalls auf dem Framework basierenden Anwendungen zurechtfinden. Zwar hat man im Grunde auch immer alle Freiheiten, eine vollkommen andere Verzeichnis- und Codestruktur zu etablieren, macht sich das Leben aber doch unnötig schwer, wie wir später noch sehen werden. Wird eine Anwendung neu gebaut, bietet es sich an, von vornherein `Zend\Mvc` einzusetzen. Will man hingegen eine bestehende Anwendung um Funktionen des Zend Framework 2 erweitern, ist es vielleicht sinnvoll, auf `Zend\Mvc` erst ganz zu verzichten oder erst zu einem späteren Zeitpunkt zu verwenden. Übrigens kann man auch das Zend Framework 2 problemlos neben Version 1 betreiben und sich beim Zend Framework 2 zunächst nur punktuell bedienen. So viel sei vorweg noch kurz gesagt.

1.1 Installation

Im Grunde muss das Zend Framework 2 gar nicht aufwändig installiert werden. Man lädt schlicht den Code herunter[1], macht ihn über einen Webserver mit PHP-Installation verfügbar und kann direkt loslegen. Eine Herausforderung ist allerdings die Tatsache, dass der Zend Framework 2 Code alleine nicht ausreicht, um die oben beschriebene, "waschechte Zend Framework 2 Anwendung" tatsächlich in Aktion zu sehen, denn wie diskutiert ist Zend\Mvc - die Komponente, die die Verarbeitung von HTTP-Requests übernimmt - optional und demnach auch nicht von Haus aus für den Einsatz "verdrahtet". Man müsste nun zunächst also selbst dafür sorgen, Zend\Mvc so mit Konfigurations- und Initialisierungs-Logik - dem sogenannten "Boilerplate-Code" - auszustatten, damit es tatsächlich auch genutzt werden kann. Andernfalls sieht man erst mal … nichts.

Um diesen Aufwand zu vermeiden und den Einstieg in die Version 2 so einfach wie möglich zu gestalten, wurde im Zuge der Entwicklung des Frameworks die sog. "ZendSkeletonApplication" erstellt, die als Vorlage für das eigene Projekt dient und den notwendigen "Boilerplate-Code" mitbringt, den man sonst aufwändig selbst erstellen müsste.

[1] http://packages.zendframework.com/

1.2 ZendSkeletonApplication

Die Installation der ZendSkeletonApplication geht am einfachsten mit Hilfe von Git. Dazu ist die Installation von Git auf dem eigenen Rechner notwendig. Auf Mac-Systemen und in vielen Linux-Distributionen ist Git sogar bereits vorinstalliert. Zur Installation auf einem Windows-System gibt es Git für Windows[2] zum Download. Die Installation unter Linux läuft fast immer über den jeweiligen Paketmanager. Nach der Installation sollte nach Aufruf von

```
1 $ git --version
```

auf der Kommandozeile dieses oder ein ähnliches Lebenszeichen zu sehen sein:

```
1 > git version 1.7.0.4
```

Ab hier ist es ganz einfach: In das Verzeichnis wechseln, in dem das Unterverzeichnis für die Anwendung angelegt werden soll und das dem Webserver später als "Document Root" zur Verfügung gestellt werden kann und die ZendSkeletonApplication herunterladen:

```
1 $ git clone git://github.com/zendframework/ZendSkeletonApplication.git
```

Okay, im Git-Jargon muss es natürlich "klonen", nicht "herunterladen" heißen. Aber da sehen wir jetzt mal drüber hinweg. Übrigens: Keine Angst vor Git! Wir nutzen Git im Rahmen dieses Buches lediglich dazu, den notwendigen Code zu beziehen, nicht für die Verwaltung unseres eigenen Anwendungscodes. Es ist also kein Git-Wissen erforderlich, dass über den Aufruf oben hinaus geht. Es steht dem Leser natürlich frei, im Weiteren den eigenen Anwendungscode dauerhaft mit Git zu verwalten - und dazu würde ich dringend raten - notwendig ist es jedoch nicht. Es kann daher problemlos später auch Subversion, CVS oder auch gar kein System zur Codeverwaltung eingesetzt werden.

Der Download der ZendSkeletonApplication geht sehr schnell, auch bei einer weniger schnellen Internetanbindung, über die man allerdings grundsätzlich auf jeden Fall verfügen muss. Der Grund für den schnellen Download ist die Tatsache, dass der Framework-Code selbst noch gar nicht heruntergeladen wurde, sondern eben nur der entsprechenden Boilerplate-Code zur Entwicklung der eigenen, auf dem Zend Framework 2 aufsetzenden Anwendung.

[2] http://code.google.com/p/msysgit/

1.3 Composer

Die `ZendSkeletonApplication` macht sich mit Composer[3] ein weiteres PHP-Tool zunutze, dass sich seit einiger Zeit für das Management von Abhängigkeiten zu anderen Code-Bibliotheken etabliert hat. Die Idee hinter Composer ist so einfach wie genial: Über eine Konfigurationsdatei wird definiert, von welchen anderen Code-Bibliotheken eine Anwendung abhängig ist und von wo die jeweilige Bibliothek bezogen werden kann. In diesem Fall ist die Anwendung abhängig vom Zend Framework 2, wie ein Blick in die Datei composer.json im Application-Root verrät:

```
1  {
2          "name": "zendframework/skeleton-application",
3          "description": "Skeleton Application for ZF2",
4          "license": "BSD-3-Clause",
5          "keywords": [
6              "framework",
7              "zf2"
8          ],
9          "homepage": "http://framework.zend.com/",
10         "require": {
11             "php": ">=5.3.3",
12             "zendframework/zendframework": "dev-master"
13         }
14 }
```

In Zeile 11 und 12 sind die beiden Abhängigkeiten, über die die Anwendung verfügt, deklariert: Es wird sowohl PHP 5.3.3 oder höher, als auch das Zend Framework 2 in einer bestimmten Version benötigt. Die folgenden 2 Aufrufe sorgen dafür, dass das Zend Framework 2 heruntergeladen und zusätzlich auch noch so in die Anwendung integriert wird, dass es sofort einsetzbar ist und die entsprechenden Framework-Klassen via Autoloading bereitgestellt werden:

```
1 $ cd ZendSkeletonApplication
2 $ php composer.phar install
3 > Installing zendframework/zendframework (dev-master)
```

Composer hat nun das Zend Framework 2 heruntergeladen und im Verzeichnis `vendor` einsatzbereit für die Anwendung verfügbar gemacht.

[3] getcomposer.org/

1.4 Ein erstes Lebenszeichen

Wir haben fast alle notwendigen Vorbereitungen getroffen. Zuletzt müssen wir nun nur noch sicherstellen, dass das Verzeichnis `public` der Anwendung als Document Root des Webservers konfiguriert und über die URL `http://localhost` über den Browser aufrufbar ist.

Etwa müsste dazu exemplarisch in der httpd.conf des Apache eine Direktive in der Form

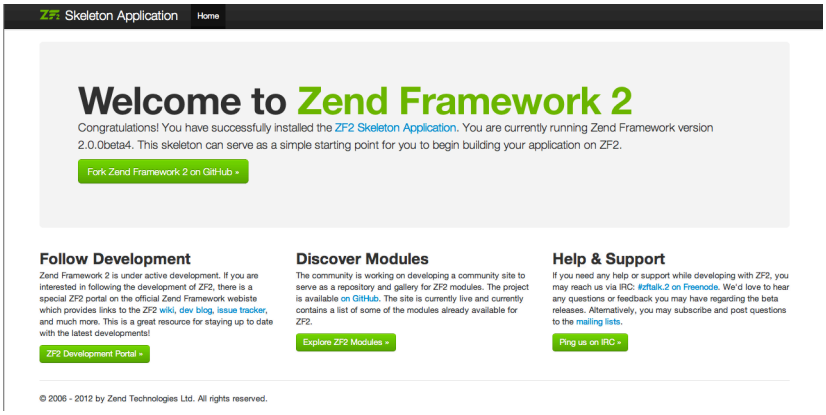
```
[..]
DocumentRoot /var/www/ZendSkeletonApplication/public
[..]
```

angegeben sein, wobei hier vorausgesetzt wird, dass die ZendSkeletonApplication zuvor mit den folgenden Kommandos heruntergeladen wurde:

```
$ cd /var/www
$ git clone git://github.com/zendframework/ZendSkeletonApplication.git
```

Ein wichtiger Hinweis an dieser Stelle: Da der Umfang des Vorwissens meiner Leser vermutlich stark unterschiedlich ist, gehe ich hier nicht darauf ein, wie genau eine Webserver samt PHP auf einem System installiert wird, sondern gehe davon aus, dass dieses Wissen bereits vorhanden ist. Für alle, die hierbei Unterstützung brauchen, findet sich im Anhang zu diesem Buch weitere Hilfestellung.

Sind die Konfigurationen gemacht, sollte sich das Zend Framework 2 das erste Mal zeigen, wenn im Browser `http://localhost` aufgerufen wird:



Startseite der ZendSkeletonApplication des Zend Framework 2

1.5 Die Datei index.php

Jeder Request, der nicht auf ein tatsächlich im Verzeichnis `public` existierendes File "mapped", wird also durch die Datei `.htaccess` auf die `index.php` umgeleitet. Sie hat daher eine besondere

Bedeutung für die Arbeit mit dem Zend Framework 2. Wichtig ist an dieser Stelle noch einmal, dass die `index.php` selbst nicht Teil des Frameworks ist, aber unbedingt gebraucht wird, um die MVC-Komponente des Frameworks einzusetzen. Wir errinnern uns: `Zend\Mvc` ist die Komponente, die den "Verarbeitungsrahmen" für eine Anwendung darstellt. Die `index.php` kommt mit der ZendSkeletonApplication und brauchen wir daher nicht selbst zu entwickeln.

Aufgrund der Wichtigkeit der `index.php` - sowohl für das Framework als auch das Verständnis für die Mechaniken des Frameworks - riskieren wir hier nun einen detaillierten Blick auf die sehr überschaubare Datei:

```
1  <?php
2  chdir(dirname(__DIR__));
3  include 'init_autoloader.php';
4  Zend\Mvc\Application::init(include 'config/application.config.php')->run();
```

Zunächst wird in das Application-Root-Verzeichnis der Anwendung gewechselt, um von dort aus auf einfache Art und Weise weitere Ressourcen referenzieren zu können. Dann wird `init_autoloader.php` aufgerufen, die zunächst das Autoloading durch Composer anstößt. Dieser unscheinbare Aufruf sorgt dafür, dass alle via Composer installierten Bibliotheken ihre Klassen über Autoloading-Mechanismen automatisch zur Verfügung stellen:

```
1  <?php
2  if (file_exists('vendor/autoload.php')) {
3          $loader = include 'vendor/autoload.php';
4  }
```

Wir können uns also sämtliche `require()` - Aufrufe in der Anwendung sparen. Was hier so emotionslos niedergeschrieben klingt, ist in Wirklichkeit eine enorme Errungenschaft für uns PHP-Entwickler: Es könnte nicht einfacher mehr sein, Bibliotheken in die eigene Anwendung zu integrieren!

In der `init_autoloader.php` wird dann alternativ noch das Autoloading der Zend Framework - Klassen über die Umgebungsvariable `ZF2_PATH` bzw. über Git Submodule sichergestellt, nur für den Fall, dass man das Zend Framework nicht über Composer bezogen hat, denn dann reicht bereits der oben genannten Autoloading-Mechanismus von Composer aus. Mit Hilfe der Umgebungsvariable `ZF2_PATH` können etwa mehrere Anwendungen auf dem System eine zentrale Installation des Framework-Codes nutzen. Ob das tatsächlich sinnvoll ist, mag ich nicht beurteilen. Dann noch ein kurzer Check, ob das Zend Framework 2 jetzt geladen werden kann - denn ohne geht's ja nicht - und dann kann es losgehen:

```
1 <?php
2 Zend\Mvc\Application::init(include 'config/application.config.php')->run();
```

Der Aufruf der Klassenmethode `init()` der `Application` sorgt zunächst dafür, dass der `ServiceManager` aufgesetzt wird. Der `ServiceManager` ist das zentrale Objekt im Zend Framework 2. Es stellt auf vielerlei Arten den Zugriff auf andere Objekte zur Verfügung, agiert meist als "Hauptansprechpartner" in der Verarbeitungskette und auch als erster Einstiegpunkt überhaupt. Wir werden uns mit dem `ServiceManager` später noch detaillierter beschäftigen. Der Einfachheit halber kann man sich den `ServiceManager` zunächst als eine Art globales Verzeichnis vorstellen, in dem zu einem definierten Key ein Objekt abgelegt werden kann. Für alle, die bereits mit Version 1 des Frameworks gearbeitet haben, stellt sich der `ServiceManager` zunächst also als eine Art Zend_Registry dar. An dieser Stelle vielleicht aber doch schon noch ein kleiner Vorgriff: Als Werte, die zu einem festgelegten Key im `ServiceManager` hinterlegt werden können, kommen nicht nur bereits erzeugte Objektinstanzen in Frage, sondern auch Factories, die die jeweiligen Objekte - im Kontext des `ServiceManager` übrigens als "Services" bezeichnet, daher der Name `ServiceManager` - erzeugen. Die Idee dahinter ist, dass diese Services erst dann erzeugt werden, wenn Sie auch tatsächlich benötigt werden. Dieses Vorgehen wird als "Lazy Loading" bezeichnet, ein Design-Pattern, um die speicher- und rechenzeitraubende Instanzierung von Objekten so lang wie möglich hinauszuzögern. Tatsächlich werden einige Services für manche Arten von Requests sogar niemals benötigt, warum sie also schon immer vorweg instanzieren?

Aber zurück zu den eigentlichen Codezeilen: Der Methode `init()` wird als Parameter die Anwendungskonfiguration übergeben und diese bereits bei der Erzeugung des `ServiceManager` berücksichtigt:

```
1 <?php
2 $serviceManager = new ServiceManager(new ServiceManagerConfig($configuration['ser\
3 vice_manager']));
```

Hier wird nun der `ServiceManager` initialisiert und mit den Services ausgestattet, die im Rahmen der Verarbeitung von Requests durch `Zend\Mvc` erforderlich sind. Das heißt also, dass der `ServiceManager` grundsätzlich auch für vollkommen andere Zwecke genutzt werden kann, auch jenseits von Zend\Mvc. Das ist wichtig zu verstehen.

Im Anschluss wird die Anwendungskonfiguration für die spätere Verwendung selbst als "Service" im `ServiceManager` abgelegt:

```
1 $serviceManager->setService('ApplicationConfig', $configuration);
```

Dann wird der `ServiceManager` bereits das erste Mal um seine Dienste gebeten:

```
$serviceManager->get('ModuleManager')->loadModules();
```

Die Methode `get()` fordert einen Service an. Hier haben wir übrigens auch schon den Fall, wo der `ServiceManager` nicht ein bereits instanziertes Objekt zurückliefert, sondern sich einer Factory bedient, um den angeforderten Service auf Zuruf zu erzeugen. Im diesem Fall kommt die `Zend\Mvc\Service\ModuleManagerFactory` zum Einsatz, die den angefragten `ModuleManager` erzeugt.

Aber woher weiß der `ServiceManager` jetzt eigentlich, dass er immer dann, wenn der Service "ModuleManager" angefragt wird, die oben genannte Factory zur Erzeugung bemüht wird? Schauen wir noch einmal in den Code davor:

```
$serviceManager = new ServiceManager(new ServiceManagerConfig($configuration['ser\
vice_manager']));
```

Durch die Übergabe der `ServiceManagerConfig` wird der `ServiceManager` auf die Nutzung mit `Zend\Mvc` vorbereitet und unter anderem ebenjene Factory für den `ModuleManager` registriert. In den nächsten Kapiteln sehen wir uns das alles auch noch einmal im Detail an und schauen auch auf die weiteren Services, die standardmäßig bereitgestellt werden.

Aber zurück zum Ablauf: Nachdem der `ModuleManager` nun also über den `ServiceManager` zur Verfügung gestellt wurde, initialisiert die Methode `loadModules()` alle registrierten Module. Auch hier verschieben wir die Details auf das nächste Kapitel, in dem wir selbst ein Modul erstellen werden. Sind die Module bereit, wird der `ServiceManager` einmal mehr gefordert und um den Service "Application" gebeten:

```
return $serviceManager->get('Application')->bootstrap();
```

Jetzt beginnt ein durchaus komplexer Vorgang, der für die eigentliche Verarbeitung des Requests zuständig ist: Die "Anwendung" wird vorbereitet ("bootstrapping"). Zurück in der `index.php` wird dann die Anwendung ausgeführt und das Ergebnis an den Aufrufer zurückgegeben:

```
Zend\Mvc\Application::init(include 'config/application.config.php')->run();
```

Der genauen Betrachtung dieses Ablaufs habe ich aufgrund seiner Wichtigkeit, aber auch der Komplexität, ein eigenes, späteres Kapitel gewidmet.

Wir halten fest: Die `index.php` ist der zentrale Einsprungspunkt für alle Requests, die durch die Anwendung verarbeitet werden. Durch die `.htaccess` werden ebenjene Anfragen technisch alle auf die `index.php` "umgebogen". Die eigentliche, vom Nutzer aufgerufene URL bleibt dabei übrigens natürlich weiter erhalten und wird später vom Framework ausgelesen, um passend ein Modul, einen Controller und eine Action aufzurufen. Der `ServiceManager` steht im Zentrum der Verarbeitung und gibt uns dann Zugriff auf die weiteren Komponenten. Daher müssen wir zunächst den `ServiceManager` erzeugen, bevor dieser uns wiederum Zugriff auf den `ModuleManager` gibt, mit dessen Hilfe wird die registrierten Module zum Leben erwecken, als auch auf die `Application`, die sich für die Verarbeitung der Requests verantwortlich fühlt. So weit, so gut.

1.6 Verzeichnisstruktur einer Zend Framework 2 Anwendung

Jetzt haben wir sie also vor uns, die "waschechte" Zend Framework 2 Anwendung, mit ihrem charakteristischen Verzeichnislayout und dem typischen Konfigurations- und Initialisierungscode:

```
application_root/
        config/
            application.config.php
            autoload/
                global.php
                local.php
                ...
        module/
        vendor/
        public/
            .htaccess
            index.php
    data/
```

Das application_root ist in unserem Fall das Verzeichnis ZendSkeletonApplication, das automatisch beim Klonen des entsprechenden GitHub-Repositories erzeugt wurde. Im Verzeichnis `config` liegt zum einen `application.config.php`, das die Basis-Konfiguration für `Zend\Mvc` und seine Kollaborateure als PHP-Array enthält. Dort werden auch zwei wichtige Manager-Komponenten des Frameworks konfiguriert: der `ServiceManager` und der `ModuleManager`, auf deren Details wir im Verlauf des Buches häufiger noch zu sprechen kommen werden. Das Verzeichnis `autoload` enthält bei Bedarf weitere Konfigurationsdaten in Form zusätzlicher PHP-Files, die zunächst etwas befremdlich wirken. Zunächst einmal irritiert die Bezeichnung "autoload" des Verzeichnisses ein wenig. Autoload hat an dieser Stelle nichts mit dem autoloading von PHP-Klassen zu tun, sondern weißt darauf hin, dass die Konfigurationen, die in diesem Verzeichnis abgelegt sind, automatisch geladen werden. Und das übrigens zeitlich nach den Konfigurationen der `application.config.php` und auch nach den Konfigurationen, die die einzelnen Module, auf die wir später noch zu sprechen kommen, vornehmen. Diese Reihenfolge der Konfigurationsauswertung ist sehr wichtig, erlaubt sie doch das situationsabhängige Überschreiben von Konfigurationswerten. Das gleiche Prinzip steht hinter `global.php` und `local.php`: Konfigurationen in der `global.php` haben immer Gültigkeit, können aber von Konfigurationen in der `local.php` überschrieben werden. Rein technisch liest das Framework zunächst die `global.php` ein und im Anschluss die `local.php`, wobei zuvor definierte Werte ggf. ersetzt werden. Wozu ist das gut? Auf diese Art und Weise können Konfigurationen abhängig von der Laufzeitumgebung definiert werden. Nehmen wir an, die Programmierer einer Anwendung haben für die Entwicklung lokal auf ihren Rechnern eine Laufzeitumgebung eingerichtet. Da für die Anwendung eine MySQL-Datenbank erforderlich ist, haben alle Entwickler diese

jeweils bei sich installiert und dabei die Zugriffsrechte so konfiguriert, dass Passwörter zum Einsatz kommen, die der jeweilige Entwickler auch sonst schon häufig verwendet. Ist ja bequemer. Da nun jeder Entwickler aber potenziell über ein individuelles Passwort für die Datenbank verfügt, lässt sich diese Konfiguration in der Anwendung nicht "hart verdrahten", sondern muss jeweils individuell angegeben werden. Dazu trägt der Entwickler individuell seine Verbindungsdaten in die Datei `local.php` ein, die er nur bei sich lokal auf dem Rechner vorhält und auch nicht in das Codeverwaltungssystem eincheckt. Während in der `global.php` die Verbindungsdaten für das "Live-System" hinterlegt sind, kann so jeder Entwickler lokal mit seinen eigenen Verbindungsdaten arbeiten, die mithilfe der `local.php` definiert sind. Auf diese Art und Weise lassen sich auch spezielle Konfigurationen für Test- oder Staging-Systeme hinterlegen.

Im Verzeichnis `module` befinden sich die einzelnen Module der Anwendung. Jedes Modul selbst kommt mit einem eigenen, typischen Verzeichnisbaum daher, das wir uns später noch genauer ansehen werden. Wichtig ist an dieser Stelle jedoch, dass jedes Modul zusätzlich eine eigene Konfiguration mitbringen kann. Wir haben mittlerweile nun 3 Stellen, an denen etwas konfiguriert werden kann: `application.config.php`, Modul-spezifische Konfiguration und die Dateien `global.php` und `local.php`, die genau in dieser Reihenfolge vom System eingelesen werden und schlussendlich ein großes, gemeinsames Konfigurations-Objekt ergeben, da im Zuge der Ausführung ebenjene Konfigurationen zusammengefügt werden. Sind die Konfigurationen der `application.config.php` nur auf den ersten Metern des Bootstrappings von Interesse, werden die Konfigurationen der Module und die von `global.php` und `local.php` auch im späteren Verlauf der Verarbeitungskette noch von Interesse sein und dankenswerter Weise durch durch `ServiceManager` zugreifbar gemacht. Auch darüber werden wir später noch mehr erfahren. Der aufmerksame Leser stellt an dieser Stelle fest, dass durch diese "Konfigurations-Kaskade" also etwa auch Modul-Konfigurationen, die im Rahmen von Modulen von Drittherstellern in die Anwendung einfließen, erweitert oder gar ersetzt werden können. Das ist sehr praktisch.

Das Verzeichnis `vendor` enthält konzeptionell all den Code, den man nicht selbst geschrieben hat (vom Code der ZendSkeletonApplication an dieser Stelle einmal abgesehen, aber den hätte man im Zweifels ja auch selbst schreiben müssen), bzw. den man nicht speziell für diese eine Anwendung geschrieben hat. Das Zend Framework 2 befindet sich also dort, aber ggf. auch weitere Bibliotheken, etwa Doctrine 2 oder Propel. Grundsätzlich muss man sich bei zusätzlichen Bibliotheken selbst darum kümmern, dass die entsprechenden Klassen aus der Anwendung heraus ansprechbar sind. Kann man die jeweilige Bibliothek aber via Composer installieren, wird einem diese Arbeit abgenommen. Die Installation weiterer Abhängigkeiten sollte im Idealfall also immer über Composer durchgeführt werden. Interessant ist auch die Tatsache, dass auch Zend Framework 2 Module, die ja eigentlich im Verzeichnis `module` ihren Platz haben, auch über das Verzeichnis `vendor` bereitgestellt werden können (korrekterweise muss man sagen, dass dies konfigurabel ist und Module quasi überall abgelegt werden können). Das heißt, dass sich auch Bibliothek von Drittherstellern, die dem Modul-Standard des Zend Framework 2 folgen, auf diese Art und Weise hinzugefügt werden können. So kann man dafür sorgen, dass sich nur die Module im Verzeichnis `module` befinden, die man tatsächlich auch selbst entwickelt hat. Alle weitere Module können auch über `vendor` bereitgestellt werden.

In `public` liegen alle Files, die über den Webserver nach außen hin zugänglich gemacht werden sollen (von spezifischen Restriktionen über die Webserver-Konfiguration einmal abgesehen). Dies ist also der Ort für Images, CSS oder JS-Files sowie den "zentralen Einstiegspunkt", der `index.php`. Die Idee hier ist diese: Jeder HTTP-Request, der den Webserver und eine bestimmte Anwendung erreicht, führt zunächst zum Aufruf der `index.php`. Immer. Egal, wie die eigentliche Aufruf-URL aussieht. Die einzige Ausnahme sind URLs, die direkt auf eine tatsächlich vorhandene Datei innerhalb oder unterhalb des Verzeichnis `public` verweisen. Nur in diesem Fall wird nicht die `index.php` zur Ausführung gebracht, sondern die entsprechende Datei gelesen und zurückgegeben. Erreicht wird dieser Mechanismus durch eine Zend Framework typische `.htaccess` - Datei im Verzeichnis `public`:

```
1 RewriteEngine On
2 RewriteCond %{REQUEST_FILENAME} -s [OR]
3 RewriteCond %{REQUEST_FILENAME} -l [OR]
4 RewriteCond %{REQUEST_FILENAME} -d
5 RewriteRule ^.*$ - [NC,L]
6 RewriteRule ^.*$ index.php [NC,L]
```

Damit dies funktioniert, müssen mehrere Bedingungen erfüllt sein. Zum Einen muss der Webserver mit einer sog. RewriteEngine[4] ausgestattet sein, die zudem aktiviert sein muss. Zum Anderen muss es der Webserver einer Anwendung erlauben, Direktiven über eine eigene `.htaccess` zu setzen. Dazu muss exemplarisch in der Apache `httpd.conf` die Direktive

```
1 AllowOverride All
```

vorhanden sein.

Das Verzeichnis `data` ist recht unspezifisch. Im Grunde genommen können hier Daten aller Art abgelegt werden, die in irgendeiner Form mit der Anwendung zu tun haben (Dokumentation, Testdaten, etc.) oder zur Laufzeit generiert werden (Caching-Daten, erzeugte Dateien, etc.).

[4] http://httpd.apache.org/docs/current/mod/mod_rewrite.html