



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

This book is for sale at <http://leanpub.com/zendframework2>

This version was published on 2013-06-01



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2012 - 2013 Michael Romer

Inhaltsverzeichnis

1	Über dieses Buch	1
1.1	Wichtige Hinweise für Amazon-Kunden	1
1.2	Die Community zum Buch	1
2	Einleitung	2
2.1	Für wen ist dieses Buch?	3
2.2	Du & ich	3
2.3	Struktur dieses Buches	3
2.4	Begleitmaterialien	4
2.5	Code Downloads	4
2.6	Wiederholungen	5
2.7	Wie du am besten mit diesem Buch arbeitest	5
2.8	Konventionen in diesem Buch	5
3	Zend Framework 2 - Ein Überblick	7
3.1	Wie das Framework entwickelt wird	7
3.2	Modul-System	9
3.3	Event-System	10
3.4	MVC-Implementierung	11
3.5	Weitere Komponenten	12
3.6	Design-Patterns: Interface, Factory, Manager & Co.	15
4	Hallo, Zend Framework 2!	18
4.1	Installation	18
4.2	ZendSkeletonApplication	19
4.3	Composer	20
4.4	Ein erstes Lebenszeichen	21
4.5	Verzeichnisstruktur einer Zend Framework 2 Anwendung	22
4.6	Die Datei index.php	25
5	Ein eigenes Modul erstellen	30
5.1	Das "Hello World" - Modul erstellen	30
5.2	Autoloading	36
6	Einmal Request und zurück	42

INHALTSVERZEICHNIS

6.1	ServiceManager	42
6.2	Einen eigenen Service schreiben	46
6.3	ModuleManager	53
6.4	Application	61
6.5	ViewManager	64
6.6	Zusammenfassung	66
7	Events	69
7.1	Einen Listener registrieren	69
7.2	Mehrere Listener gleichzeitig registrieren	70
7.3	Einen registrierten Listener entfernen	72
7.4	Ein Event auslösen	72
7.5	SharedEventManager	74
7.6	Events in eigenen Klassen verwenden	75
7.7	Das Event-Objekt	82
8	Module	85
8.1	Das Modul "Application"	85
8.2	Modulabhängiges Verhalten	88
8.3	Ein Fremdmodul installieren	89
8.4	Ein Fremdmodul konfigurieren	92
9	Controller	96
9.1	Konzept & Funktionsweise	96
9.2	Controller Plugins	97
9.3	Ein eigenes Controller Plugin schreiben	102
10	Views	105
10.1	Konzept & Funktionsweise	105
10.2	Layouts	107
10.3	View Helper	108
10.4	Einen eigenen View Helper schreiben	108
11	Model	110
11.1	Entities, Repositories & Value Objects	111
11.2	Business-Services & Factories	113
11.3	Business-Events	114
12	Routing	115
12.1	Einleitung	115
12.2	Definition von Routen	115
12.3	Test auf Übereinstimmung	117
12.4	Erzeugung von URLs	123
12.5	Standard-Routing	124

12.6	Kreatives Routing: A/B-Tests	126
13	Dependency Injection	128
13.1	Einführung	128
13.2	Zend\Di für Objektgraphen	133
13.3	Zend\Di für das Konfigurationsmanagement	142
14	Persistenz mit Zend\Db	144
14.1	Datenbanken anbinden	146
14.2	SQL-Statements erzeugen & ausführen	147
14.3	Mit Tabellen und Entities arbeiten	150
14.4	Organisation von Datenbankabfragen	155
14.5	Zend\DB-Alternative: Doctrine 2 ORM	165
15	Validatoren	166
15.1	Standard-Validatoren	166
15.2	Eigene Validatoren schreiben	168
16	Webforms	170
16.1	Eine Form erstellen	170
16.2	Eine Form anzeigen	173
16.3	Formulareingaben verarbeiten	174
16.4	Formulareingaben validieren	176
16.5	Standard-Formelemente	183
16.6	Fieldsets	185
16.7	Entities mit Forms verbinden	190
17	Logging & Debugging	203
17.1	Einleitung	203
17.2	FirePHP für Firefox	204
17.3	ChromePHP für Chrome	205
17.4	Zend\Log	206
17.5	Zend\Log mit FirePHP	207
17.6	Zend\Log mit ChromePHP	208
17.7	Zend\Log situationsabhängig einsetzen	209
17.8	Weitere Möglichkeiten mit Zend\Log	214
18	Unit Testing	215
18.1	Einleitung	215
18.2	Einen Service testen	216
18.3	Einen Controller testen	223
19	Benutzerverwaltung	235
19.1	Einleitung	235
19.2	Nutzer-Authentifizierung	235

INHALTSVERZEICHNIS

19.3	Nutzer-Sessions	247
19.4	Ressourcen & Rollen	251
19.5	ZfcUser	255
20	ZF2 auf der Kommandozeile	257
20.1	Einleitung	257
20.2	Aufruf über die Konsole	257
20.3	Console-Routen und Request-Verarbeitung	261
21	Internationalisierung	266
21.1	Zahlen-, Währungs- und Datumsformate	267
21.2	Textübersetzungen	271
21.3	Ermitteln und Konfigurieren der Locale	277
22	AJAX & Webservices	279
22.1	AJAX mit JSON	279
22.2	AJAX mit HTML	280
22.3	RESTful-Webservices	282
23	E-Mails versenden	287
23.1	Eine E-Mail versenden	287
23.2	Eine HTML-E-Mail versenden	288
23.3	Templates für HTML-E-Mails einsetzen	288
23.4	E-Mails mit Anhängen versenden	291
24	Menüs & Co.	292
24.1	Navigationen erzeugen	292
24.2	Eine Anwendungsnavigation erzeugen	297
24.3	Breadcrumbs erzeugen	299
25	Anwendungen durch Caching beschleunigen	300
25.1	Einleitung	300
25.2	Opcode-Cache	300
25.3	Zend\Cache	301
25.4	Performance-Flaschenhälse identifizieren	304
26	Entwicklertagebuch (Praxisteil)	306
26.1	Einführung	306
26.2	Envisioning	306
26.3	Sprint 1 - Code-Repository, Entwicklungsumgebung, initiale Codebase	307
26.4	Sprint 2 - Modul anlegen, Produkte eintragen	310
26.5	Sprint 3 - Deal anlegen, Deals anzeigen	342
26.6	Sprint 4 - Bestellformular	368
26.7	Sprint 5 - Modul auslagern	372
26.8	Ideen für die Weiterentwicklung	382

INHALTSVERZEICHNIS

27 LAMP-Umgebung einrichten	384
27.1 Virtuelle Maschine generieren lassen	384

1 Über dieses Buch

1.1 Wichtige Hinweise für Amazon-Kunden

Wenn du dieses Buch bei Amazon erworben hast, ist es derzeit etwas schwieriger sicherzustellen, dass du alle Updates des Buches automatisch bekommst. Um Probleme zu vermeiden, sende mir bitte nach dem Kauf eine kurze E-Mail mit dem Betreff "Updates" an zf2buch@michael-romer.de. So kannst du sicher sein, dass du tatsächlich immer den aktuellen Stand dieses Buches vorliegen hast. Als kleines Dankeschön für deine Mühen bekommst du zusätzlich dann auch immer die PDF- und EPUB-Versionen des Buches.

1.2 Die Community zum Buch

Du hast einen Fehler gefunden, möchtest deine Ideen für die nächste Auflage loswerden oder dich einfach nur zu den Themen im Buch und zum Zend Framework 2 im Allgemeinen austauschen? Dann ist die [Community zum Buch](https://groups.google.com/forum/#!forum/webentwicklung-mit-zend-framework-2)¹ bei Google Groups der richtige Anlaufpunkt für dich. Dort bekommst du nicht nur direkten Kontakt zu mir, dem Autor dieses Buches, sondern auch zu den anderen Lesern. Du benötigst lediglich einen Google-Account, den du dir bei Bedarf zuvor kostenlos erstellst.

¹<https://groups.google.com/forum/#!forum/webentwicklung-mit-zend-framework-2>

2 Einleitung

Das Zend Framework 2 ist ein Open Source Framework zur Entwicklung professioneller Webanwendungen ab PHP 5.3. Es wird auf dem Server betrieben und dient vor allem dazu, dynamische Webinhalte zu erzeugen, oder Transaktionen, etwa den Kauf eines Produktes in einem Online-Shop, abzuwickeln. Das Zend Framework 2 kann dabei bei der Entwicklung jeglicher Art von (Web-)Anwendung unterstützen, stellt es doch allgemeingültige Lösungen bereit, die sich etwa sowohl in E-Commerce-, Content-, Community- oder SaaS-Anwendungen gleichermaßen einsetzen lassen. Das Zend Framework wird maßgeblich von der Firma Zend entwickelt, die auch als Namensgeber für das Framework fungiert und ihm ein solides (finanzielles) Fundament gibt, nicht zuletzt auch deshalb, weil Zend an der Entwicklung der Programmiersprache PHP selbst beteiligt ist. Neben Zend unterstützen aber auch noch eine Reihe weiterer namhafter Firmen das Zend Framework und sind an dessen dauerhaftem Erfolg interessiert. Dazu zählen auch Microsoft und IBM. All das gibt bei der Wahl für das Zend Framework Sicherheit und macht es auch aus betriebswirtschaftlicher Sicht zu einer guten Entscheidung für die Basis der eigenen Anwendung.

Das Zend Framework hat am 30.06.2007 in der stabilen Version 1.0 erstmals das Licht der Welt erblickt und seitdem das Erscheinungsbild der Webentwicklung mit PHP nachhaltig geprägt. Man kann wohl sagen, dass die Programmierung mit PHP durch das Zend Framework auch für unternehmenskritische Anwendung erst so richtig salonfähig geworden ist. Hat man in der Vergangenheit im Unternehmenskontext sonst doch lieber auf komplexe J2E-Anwendungen (Java) gesetzt, so wird mittlerweile auch gerne und mit gutem Gewissen zu PHP und dem Zend Framework gegriffen, bietet diese Kombination doch ein ausgewogenes Verhältnis von Leichtgewichtigkeit und Professionalität, wie kaum eine andere Plattform.

Berücksichtigt man auch die ersten Preview Releases des Zend Framework 1, so ist das Framework heute bereits mehr als 6 Jahre alt. Auch wenn sich in den unzähligen Releases seit der ersten Preview bis hin zur aktuellen Version 1.11 bereits viel getan hat, so sind einige dringend notwendige Verbesserungen und Erweiterungen nun nicht mehr auf der alten Codebasis machbar und rechtfertigen ein neues Major Release, dass erstmals nicht mehr kompatibel zu früheren Versionen ist. Es ist Zeit für einen Neuanfang.

Das Zend Framework 2 markiert den nächsten Meilenstein in der Evolution der PHP-Web-Frameworks, aber auch von PHP selbst, denn wie man nicht zuletzt an Java oder Ruby sieht, reicht eine gute Programmiersprache alleine nicht aus, um tatsächlich auch in der Breite erfolgreich zu sein. Erst mit auf der Programmiersprache aufsetzenden Frameworks wie Struts, Spring, Rails oder eben auch dem Zend Framework, die signifikant den initialen Entwicklungs- aber auch den dauerhaften Wartungsaufwand einer Anwendung senken, kriegt eine Webentwicklungs-Plattform wirklich Traktion.

2.1 Für wen ist dieses Buch?

Es ist eine Herausforderung, ein technisches Buch zu schreiben, dass die Balance zwischen Theorie und Praxis findet und es Einsteigern sowie Profis gleichermaßen ermöglicht, das Beste aus dem Buch herauszuholen. Dieser Aufgabe stelle ich mich gerne, erlaube mir allerdings ein Hintertürchen: Wenn ich das Gefühl habe, dass wir uns in Details verlieren, die an der jeweiligen Stelle oder gar ganz in diesem Buch nicht genauer erläutert werden können, verweise ich auf spätere Passagen im Buch oder auf Sekundärquellen.

Eine weitere Herausforderung ist etwaiges Zend Framework-Vorwissen bei meinen Lesern. Ein Entwickler, der erst mit der Version 2 in das Zend Framework einsteigt, benötigt an der ein oder andere Stelle andere Informationen, als ein "alter Hase", der sich in Version 2 an vielen Ecken schnell zurechtfindet, weil er die Ideen und Konzepte noch aus Version 1 kennt. Wenn ich es für sinnvoll erachtet, werde ich für alle, die Wissen zur Version 1 haben, an den entsprechenden Stellen immer wieder mal auf den Vorgänger der Version 2 referenzieren, ohne aber zu sehr ins Detail zu gehen. Neulingen mag auch das vielleicht sogar helfen, kriegen sie so doch ein besseres Gefühl dafür, warum die Version 2 notwendig ist. Dieses Buch soll sowohl für Zend Framework-Anfänger, als auch für Fortgeschrittene hilfreich sein.

Ich setze voraus, dass du über Grundkenntnisse zu PHP verfügst. Du musst kein PHP-Experte sein, vor allem nicht, weil viele "native" PHP-Funktionen sogar mit der Nutzung des Frameworks obsolet werden, wie etwa das Session-Management, das objektorientiert abgebildet und dadurch einiges an Low-Level-Funktionalität dankenswerter Weise abstrahiert wird. Wenn du dich also mit der PHP-Syntax wohlfühlst, ein Grundverständnis für die Arbeitsweise von PHP-Anwendungen hast und dir die gängigen Funktionen des Sprachkerns geläufig sind, bist du gut gerüstet. Notfalls nimmst du dir halt ein PHP-Buch dazu.

2.2 Du & ich

Hallo, ich bin Michael. Ich hoffe, es ist für dich okay, wenn ich dich in diesem Buch durchweg duze. Das macht mir das Schreiben einfacher und sorgt für eine weniger formale Atmosphäre.

2.3 Struktur dieses Buches

Dieses Buch soll kein Kompendium werden, sondern eine pragmatische und praxisorientierte Einführung in die grundlegenden Konzepte und die praktische Arbeit mit dem Zend Framework 2. Als Kompendium dienen dem Entwickler ab einem gewissen Punkt die offiziellen Dokumentationen, die aber nicht optimal dazu geeignet sind, um sich mit dem Thema vertraut zu machen, sondern viel mehr als (unverzichtbares!) Nachschlagewerk für weitere Detailfragen dienen, nachdem man sich einmal das nötige Grundverständnis erarbeitet hat. Und genau das ist Ziel dieses Buches.

Es ist so aufgebaut, dass du es von vorne bis hinten hintereinander weg lesen kannst und auch solltest. Wir starten mit einem Überblick über das Framework und schauen uns die maßgeblichen Konzepte und Ideen an, die die Gestalt des Frameworks prägen und es auch von seinem Vorgänger differenzieren. Auf dem Weg blicken wir immer wieder auch nach links und rechts und machen uns mit einigen Rahmenbedingungen vertraut, etwa die Art und Weise, wie das Framework eigentlich entwickelt wird. Dann werden wir detaillierter und durchleuchten die wichtigsten Komponenten und Zusammenhänge des Frameworks, schauen uns an, wie die Verarbeitung eines HTTP-Requests funktioniert und schreiben den ersten Code. Es folgt ein Ausflug in das Ecosystem des Frameworks. Wir blicken auf die wichtigsten Module, die von Dritten zur Verfügung gestellt werden und etwa die Realisierung einer eigenen Nutzerverwaltung obsolet machen. In dem Konzept der Module steckt ein Großteil der Magie der neuen Version des Frameworks und Module können - wie wir später noch sehen werden - die Entwicklung eigener Anwendungen massiv beschleunigen und vereinfachen. Es ist zu erwarten, dass in kurzer Zeit eine große Anzahl hochqualitativer Module für das Zend Framework 2 zu haben sein wird.

Last but not least kommt ein zusätzlicher, intensiver Praxisteil in Form eines “Entwicklertagebuchs”. Wir werden gemeinsam eine Webanwendung entwickeln, die im Anschluss auch tatsächlich in einen ernsthaften Betrieb übergehen soll (wir geben uns also besser ordentlich Mühe!). Spätestens hier wird es also richtig “hands-on”.

2.4 Begleitmaterialien

Auf der [Homepage zum Buch¹](#) findet sich das [Zend Framework 2 Cheat Sheet²](#) und weiteres Material, das zum Verständnis des Frameworks beiträgt. Es kann dort eingesehen oder heruntergeladen werden.

2.5 Code Downloads

Es existieren drei Code Repositories bei GitHub, in denen Code Listings des Buches als Referenz zum Download bereitstehen:

- [Codebeispiele aus Teil 1 des Buches³](#)
- [Demo Anwendung aus Teil 2 des Buches⁴](#)
- [Demo Modul aus Teil 2 des Buches⁵](#)

¹<http://zendframework2.de>

²<http://zendframework2.de/cheat-sheet.html>

³<https://github.com/michael-romer/zf2book>

⁴<https://github.com/michael-romer/ZfDealsApp>

⁵<https://github.com/michael-romer/ZfDeals>

Einige Code Listings wurden etwas angepasst, damit sie im Rahmen des jeweiligen Repositories funktionieren. Bei Code Listings, die im Buch Stück für Stück entwickelt werden, findet sich im Repository in der Regel jeweils nur die fertige Version.

2.6 Wiederholungen

Du wirst feststellen, dass ich in diesem Buch öfters Sachverhalte mehrfach erkläre. Das liegt entweder daran, dass ich den Überblick darüber verloren habe, was ich eigentlich schon gesagt habe und was nicht (-D) oder aber ich tue es bewusst. Denn schon die alten Römer wussten: Repetitio mater studiorum est – Wiederholung ist die Mutter des Lernens. Aber auch die Tatsache, dass der jeweilige Sachverhalte jeweils in einem anderen Kontext und damit aus einer anderen Blickrichtung diskutiert wird, ist für das Verständnis förderlich. Falls du also mal über eine Stelle im Buch stolperst, bei der du dir denkst “Weiß ich doch alles schon!”, fühle dich bitte einfach bestätigt in deinem Wissen und fahre entspannt fort oder überspringe die jeweilige Passage.

2.7 Wie du am besten mit diesem Buch arbeitest

Programmieren (oder den Umgang mit einem Programmier-Framework) lernt man am besten, wenn man selbst aktiv wird. Zwar werden insbesondere die ersten Kapitel bis zum Praxisteil auch ohne geöffnete IDE schon hilfreich sein, den größtmöglichen Lernerfolg wirst du aber haben, wenn du die ein oder andere Codezeile selbst nachvollziehst oder selbst schreibst. Im Idealfall hast du ein System mit einem Debugger zur Verfügung, mit dem du die Arbeitsweise des Frameworks Schritt für Schritt selbst nachvollziehen kannst. Ein paar Grundkenntnisse zu Git und ein github-Account können ebenfalls nicht schaden, sind aber nicht essentiell.

2.8 Konventionen in diesem Buch

Code-Beispiele

Die Listings in diesem Buch verfügen wenn möglich über Syntax-Highlighting, halten sich zwecks besserer Lesbarkeit aber nicht an immer einen Coding-Standard. PHP-Code wird durch ein `<?php` eingeleitet, ein `// [. .]` weist in dem jeweiligen Listing auf ausgesparte Code-Fragmente hin.

Viele Listings verfügen über einen Link zu github, wo sie in Form ein sog. “gist” einfach heruntergeladen oder via “Copy & Paste” übernommen werden können. So lassen sich die Codebeispiele einfach auf dem eigenen System nachvollziehen.

Kommandozeile

Wenn auf der Kommandozeile ein Kommando ausgeführt werden muss, so wird dies durch ein vorangestelltes Dollarzeichen (\$) symbolisiert. Das visuelle Feedback des Kommandos wird durch ein vorangestelltes “Größer-Zeichen” angezeigt. Beispiel:

```
1 $ phpunit --version
2 > PHPUnit 3.6.12 by Sebastian Bergmann.
```

So, jetzt aber genug der Vorrede. Legen wir los!

3 Zend Framework 2 - Ein Überblick

Bevor wir im weiteren Verlauf des Buches in die Details des Frameworks eintauchen, wollen wir uns zunächst einen Überblick verschaffen und einige Kernaspekte und Gedanken des Frameworks beleuchten. Welche Haupteigenschaften machen das Framework also aus?

3.1 Wie das Framework entwickelt wird

Das Zend Framework 2 ist Open Source. Das heißt zunächst einmal, dass der Quellcode für jedermann einsehbar und für die eigenen Zwecke nutzbar ist. Das Framework wird unter der “New BSD License” entwickelt. War es unter der ursprünglichen “BSD License” noch erforderlich, an den entsprechenden Stellen in der eigenen Anwendung auf die Nutzung einer unter der “BSD Licence” stehenden Bibliothek / eines Frameworks hinzuweisen, so ist dies beim Zend Framework nicht der Fall; die sog. “Marketing-Klausel” gibt es in der “New BSD License” nicht. Wir wollen nicht in das Software-Lizenzrecht abdriften, aber grundsätzlich an dieser Stelle kurz festhalten, dass es das Framework in rechtlicher Hinsicht wirklich gut mit uns meint, gehört die “New BSD License” doch zu den sog. “Free Software Licenses”, die so gut wie keine Limitationen oder Vorgaben für die Verwendung des Codes mit sich bringen.

Wurde während der Version 1 noch SVN für die Codeverwaltung eingesetzt, hat sich das Projekt-Team für das neue Major Release dazu entschlossen, den [Framework-Code auf github¹](https://github.com/zendframework/zf2) zu verwalten. Github unterstützt sehr gut die gemeinsame, verteilte Arbeit an der Codebase, sind doch mittlerweile viele Programmierer an der Entwicklung beteiligt und über den Globus verteilt. So ist etwa Rob Allen in England beheimatet, während Matthew Weier O’Phinney, Project Lead für das Zend Framework 2, in den USA und Ben Scholzen in Deutschland lebt. Die Genannten sind übrigens alle sog. “Core Contributor”, haben also bereits einen signifikanten Beitrag zum Framework geleistet und nehmen daher eine Sonderstellung im Team ein. Sie prägen maßgeblich die Gestalt des Frameworks. Einige der Programmierer sind direkt bei Zend angestellt, so etwa Matthew Weier O’Phinney, andere sind als Freiberufler tätig oder stehen in anderen Arbeitsverhältnissen, die eine Mitarbeit beim Zend Framework erlauben - während oder nach der Arbeitszeit. Ein bunter Haufen von guten Leuten also. Die Komponentenorientierung des Zend Framework ermöglicht den Einsatz sog. “[Component Maintainers²](http://framework.zend.com/wiki/display/ZFDEV2/Component+Maintainers)”, die den Hut für eine bestimmte Komponente des Frameworks auf haben und selbst sowie gemeinsam mit weiteren Programmierern die Geschicke einer bestimmten Komponente lenken.

Das Team organisiert sich vornehmlich über [Wiki³](http://framework.zend.com/wiki/dashboard.action), Mailinglisten und IRC-Chats. Der “Zend Framework Proposals Process” bietet allen die Möglichkeit, Vorschläge für die Weiterentwicklung

¹<https://github.com/zendframework/zf2>

²<http://framework.zend.com/wiki/display/ZFDEV2/Component+Maintainers>

³<http://framework.zend.com/wiki/dashboard.action>

des Frameworks einzureichen und regelmäßige “Bug Hunting Days” helfen, bekannte Probleme im Framework mit gemeinsamer Kraft fokussiert zu beheben. Regelmäßig werden neue Versionen des Frameworks veröffentlicht, mit denen Bugfixes und neue Features verfügbar gemacht werden.

Der PSR-2 Coding-Standard

Der Code des Frameworks selbst wird über alle Komponenten hinweg unter Berücksichtigung des [PSR-2 Coding-Standards](#)⁴ entwickelt. Die Idee des “PHP Specification Request”, kurz “PSR”, ist inspiriert vom [Java Specification Request](#)⁵. Über dieses Verfahren werden neue Java-Standards definiert und Erweiterungen der Programmiersprache Java oder der Java-Laufzeitumgebung gemeinschaftlich entwickelt und herstellerübergreifend vereinbart. Dieses Vorgehen hat für Anwendungsentwickler viele Vorteile, macht es die eigentliche Anwendung doch deutlich portabler und herstellerunabhängig. So kann auf (mehr oder weniger) einfache Art und Weise etwa der Anbieter des eigenen Application-Servers gewechselt werden.

Eine ähnliche Idee liegt auch den “PHP Specification Requests” zugrunde, sollen sie doch insbesondere dafür sorgen, dass Softwarekomponenten unterschiedlicher Hersteller und Frameworks kompatibel zueinander sind und kombiniert eingesetzt werden können. Im Gegensatz zum “JSR” ist das “PSR”-Vorgehen noch recht neu. Bislang hat man sich auch erst auf drei Spezifikationen verständigt:

- PSR-0: Definiert den Zusammenhang zwischen PHP-Namespace und der Organisation von PHP-Dateien in einem Dateisystem, um das Autoloading von Klassen, auch komponenten- und herstellerübergreifend, so einfach wie möglich zu gestalten.
- PSR-1 / PSR-1-basic: Ein neuer, gemeinsamer Coding-Standard.
- PSR-2: Eine Erweiterung des PSR-1-Coding-Standards.

Ein Hauptaugenmerk der Version 2 des Zend Framework liegt, wie wir im Verlauf dieses Buches auch immer wieder sehen werden, auf der funktionalen Erweiterung einer Anwendung durch wiederverwendbare und einfach zu integrierende Module. Die Einhaltung der PSR-Standards ist dabei eine enorme Hilfe.

Bekannte Probleme

Wie jede größere Software ist auch das Framework nicht ganz frei von Fehlern. Konsequenterweise kommt zur Nachverfolgung [github issues](#)⁶ zum Einsatz. Wenn du also ein Problem feststellst, lohnt es sich, zunächst dort zu schauen, ob der Fehler bereits bekannt ist. Falls nicht, kannst du dort ein Ticket öffnen.

⁴<https://github.com/pmjones/fig-standards/blob/psr-1-style-guide/proposed/PSR-2-advanced.md>

⁵http://de.wikipedia.org/wiki/Java_Specification_Request

⁶<https://github.com/zendframework/zf2/issues?state=open>

3.2 Modul-System

Das Modul-System ist zentraler Dreh- und Angelpunkt der Version 2. Matthew Weier O'Phinney sagte es in der Mailing-List mehr als deutlich:

“Modules are perhaps the most important new development in ZF2.”

Für Version 2 wurde das Modul-System des Frameworks vollständig überarbeitet. War es zwar in Version 1 bereits möglich, den eigenen Code in Modulen zu organisieren, so waren diese Module tatsächlich aber nie eigenständig nutz- oder auf andere Anwendungen übertragbar, ohne das man das Modul dort aufwendig neu in den jeweiligen Code hätte einweben müssen. Der Aufwand dafür war meist so groß, dass man die Funktion auch gleich selbst bauen konnte. Das Modul-System der Version 1 war demnach auf die Vorteile einer besseren Codeorganisation innerhalb einer abgeschlossenen Anwendung begrenzt. Das hat dazu geführt, dass wir jetzt auf der Welt vermutlich tausende von Implementierungen für die Authentifizierung eines Nutzers haben, oder ähnliche Funktionen, die allgemein gültig, weil nicht auf eine konkrete Anwendung begrenzt, sind.

Von Anwendungen wie WordPress, Drupal oder Magento ist man es gewohnt, funktionale Erweiterungen in Form von Plugins oder Extensions hinzuzufügen. Auch [Symfony 2⁷](#) - ein populäres, alternatives Webframework - verfügt mit seinen Bundles bereits seit einiger Zeit über ein Modul-Konzept, dass es beispielsweise ermöglicht, die Funktionalität eines Content Management Systems (CMS) in die eigene Anwendung zu integrieren, ohne selbst etwas programmieren zu müssen. Und nun hat auch das Zend Framework in der neuen Version diese Erweiterungsmöglichkeit erhalten. Nehmen wir noch einmal ein konkretes Beispiel: Eine auf Basis des Zend Frameworks entwickelte Anwendung benötigt im weiteren Verlauf zusätzlich die Funktionalität eines Blogs. Eine geläufige Anforderung, ist ein Blog doch etwa als SEO-Maßnahme äußert sinnvoll. Wer schon einmal mit Wordpress & Co. gearbeitet hat, der weiß, wie umfassend die Anforderungen an ein modernes Blog-System mittlerweile sind. Es wird schnell klar, dass es keine gute Idee ist, jetzt selbst eine eigene Blogging-Software zu entwickeln.

Stattdessen erscheint es sinnvoll, eines der verfügbaren freien oder kommerziellen Blogging-Systeme einzusetzen, die aber konzeptbedingt nur als Insellösung “neben” die eigene Anwendung gestellt werden können. Dies hat einige Nachteile: Etwa kann das Logging-System der eigenen Anwendung hier nicht ohne Weiteres zum Einsatz kommen. Der Caching-Layer auch nicht. Die Daten des Blogs liegen in einer anderen Datenbank und wenn wir also die letzten 3 Blog-Post-Teaser auf der Homepage unserer Anwendung anzeigen wollen, müssen wir diese Einbindung wohl über eine API des Blogs, eventuell über einen RSS-Feed oder ähnliches, realisieren (oder in der Datenbank einer fremden Anwendung herumpfuschen ...). Die Administratoren-Accounts, die es in unserer Anwendung den Mitarbeitern unseres Unternehmens erlauben, Kundenstammdaten zu verwalten, existieren in dem Blog-System so natürlich auch erstmal nicht, von [Single-Sign-On⁸](#) ganz zu schweigen. Unser Corporate-Design müssen wir im Template-System des Blogs noch

⁷<http://symfony.com/>

⁸http://de.wikipedia.org/wiki/Single_Sign-on

einmal nachbauen, weil uns Layout, Markup und Styles nicht ohne Weiteres dort zur Verfügung stehen. Etwaige Design-Anpassungen müssen wir zukünftig auch dort immer nachziehen. Die Build-Skripte unserer Anwendungen können wir für den Blog nicht einsetzen, die Release-Prozesse müssen wir dahingehend anpassen, das wir irgendwie auch das fremde Blog-System mit einschließen können. Wir schlittern mit diesem Ansatz also direkt in die Komplexität der [Enterprise Application Integration](#)⁹ (EAI) und [Service Oriented Architecture](#)¹⁰ (SOA) und schaffen uns so einen bunten Strauß neuer Probleme und Herausforderungen. Stünde uns stattdessen ein Blog-Modul für das Zend Framework 2 zur Verfügung, das sich nahtlos in die bestehenden Authentifizierungs-, Build & Release-, Caching-, Logging- und Design-Implementierungen einfügen würde, wäre alles viel einfacher, ja nahezu trivial. Und genau dieser Gedankengang ist die Kernidee des Modul-Systems.

Ein Zend Framework Modul bringt alles mit, was es für seinen Betrieb benötigt. Dazu gehört nicht nur der entsprechende PHP-Code, sondern auch HTML-Templates, CSS, JavaScript-Code, Images und so weiter, so das ein Modul ein echtes, in sich abgeschlossenes Paket ergibt. Ein gutes Modul lässt sich mit wenigen Handgriffen in eine Zend Framework 2 Anwendung integrieren und ... läuft einfach. Das Modul-System macht das Zend Framework 2 also zu viel mehr als einem Web-Framework, sondern darüber hinaus zu einer Plattform für integrierte Anwendungen und Funktionen.

Im nächsten Kapitel gehen wir auf die technischen Details des Modulsystems ein und schauen uns an, wie es intern arbeitet und wie Module entwickelt werden, denn auch die eigene Anwendung wird im Code durch ein Modul repräsentiert. Module sind überall im neuen Zend Framework! Das Verständnis für Module ist also die halbe Miete, sowohl für die Entwicklung der eigenen Anwendung, als auch für die Einbindung bereits implementierter Funktionen und Systeme.

Im späteren Verlauf des Buches werfen wir dann auch einen Blick auf verfügbare Module, denn neben den bereits angesprochenen Funktionen für das Usermanagement gibt es noch eine ganze Menge mehr, etwa Module rund um [Doctrine](#) ²¹¹, das bekannte PHP-[ORM](#)¹²-System. Bei diesen Modulen handelt es sich um sog. "Glue Code", der es erlaubt, diese Bibliothek auf einfache Art und Weise in einer Zend Framework 2 Anwendung einzusetzen.

3.3 Event-System

Das Zend Framework 2 basiert maßgeblich auf dem Konzept der [Event-driven Architecture](#)¹³. Diesem Ansatz liegt die Idee zugrunde, dass gewisse Aktivitäten in einem System stattfinden, nachdem zuvor ein bestimmtes Ereignis eingetreten ist. Dazu registrieren sich Aktivitäten für die Benachrichtigung bei Eintritt des Ereignisses. Tritt das Ereignis ein, werden die registrierten Aktivitäten ausgeführt. Eine Analogie aus der realen Welt: Wenn wir an der Haltestelle warten (wir haben uns für das Ereignis des eintreffenden Busses "registriert") und der Bus dann endlich vorfährt

⁹http://de.wikipedia.org/wiki/Enterprise_Application_Integration

¹⁰http://de.wikipedia.org/wiki/Serviceorientierte_Architektur

¹¹<http://www.doctrine-project.org/>

¹²http://de.wikipedia.org/wiki/Objektrelationale_Abbildung

¹³http://en.wikipedia.org/wiki/Event-driven_architecture

(Ereignis tritt ein), dann öffnet sich die Tür (Aktivität 1), kaufen wir ein Ticket (Aktivität 2), suchen uns einen freien Platz (Aktivität 3) und der Bus fährt los (Aktivität 4).

Ein weiteres Beispiel: Ein bei eBay angebotener Artikel wird verkauft. Dieses Ereignis löst eine Reihe von Aktivitäten im System aus:

- Es wird eine Kaufbestätigung an den Käufer gesendet.
- Es wird eine Verkaufsbestätigung an den Verkäufer gesendet.
- Es werden die Gebühren für den Verkäufer kalkuliert und dem Verkäuferkonto zugerechnet.
- Der betreffende Artikel wird aus dem Suchindex entfernt, damit er nicht mehr gefunden wird (er ist ja bereits verkauft).

Entschließt sich eBay zu einem späteren Zeitpunkt dazu, bei Verkauf eines Artikels auch die unterlegenen Bieter über das Ende der Auktion zu informieren (in Wirklichkeit tun sie das natürlich bereits), lässt sich diese Aktivität jederzeit zusätzlich zu den anderen an dieses Ereignis hängen.

EDA ist allerdings ein zweischneidiges Schwert. Auf der einen Seite gewinnt man durch diesen Architekturstil eine enorme Flexibilität in der Gestaltung von Workflows. Neue Aktivitäten lassen sich einfach hinzufügen. Ganze Abläufe können im Nachgang auf einfache Art und Weise modifiziert werden. Flexibilität ist das schlagende Argument. Dem gegenüber steht eine gewisse Intransparenz über die Dinge, die da eigentlich alle so in der Anwendung passieren, wenn ein Ereignis eintritt. Grundsätzlich kann mehr oder weniger überall in der Anwendung eine Aktivität für ein Ereignis registriert werden, ohne, das an der Stelle im Code, an der das Ereignis später tatsächlich auftritt, diese Verbindung ersichtlich wäre. Eine weitere Herausforderung ist die Reihenfolge der Aktivitäten: Soll der Verkäufer in dem eBay-Beispiel von oben etwa in der Verkaufsbestätigung auch schon über die angefallenen Gebühren informiert werden (die sich auf Basis des Endpreises der Aktion berechnen), muss diese Aktivität stattfinden, nachdem zuvor die Gebühren kalkuliert wurden (also eine andere Aktivität ausgeführt wurde). Hier wird es also langsam kompliziert. Kurz gesagt: EDA kann zu schwer durchschaubaren Abläufen führen. Fehlerursachen lassen sich schwieriger identifizieren, Anwendungen sind aufwändiger zu debuggen.

Dennoch überwiegen die Vorteile so deutlich, insbesondere auch in Verbindung mit dem Modulsystem des Frameworks, so dass man sich beim Zend Framework 2 für EDA entschieden hat. Kennt man die Fallstricke, kann man sie leicht umgehen.

3.4 MVC-Implementierung

Auch im Zend Framework 2 steht die Implementierung des MVC-Patterns im Zentrum des Frameworks. Zwar bietet auch das Zend Framework 2 wieder die Möglichkeit, sich frei bei dessen Komponenten zu bedienen und etwa `Zend\Mvc` links liegen zu lassen, praktisch wird man dies aber doch nur in Ausnahmefällen tun. Ist es doch in aller Regel gerade die MVC-Implementierung und die sich daraus ergebende vorteilhafte Codestruktur der eigenen Anwendung, die den Griff zum Zend

Framework oftmals begründet. Zend\Mvc strukturiert eine Anwendung durch die logische Trennung von Codebestandteilen in Models, Views und Controller und sorgt auf diesem Weg für eine gewisse Ordnung, die nicht nur der Wart- und Erweiterbarkeit der Anwendung zuträglich ist, sondern auch die Wiederverwendung von Funktionen fördert.

Und so sieht Zend\Mvc in Aktion aus: Nachdem der Zend\ModuleManager, die zentralen Einheit im Modul-System, alle verfügbaren Module für den Einsatz vorbereitet hat, die Konfiguration eingelesen und weitere Komponenten initialisiert sind, sorgt der Zend\Mvc\Router dafür, dass ein passender Controller (eine Klasse) in einem Modul instanziiert und die richtige Action (eine Methode) darin aufgerufen wird. Das Routing basiert dabei auf zuvor konfigurierten Routen, die das Mapping zwischen URLs und Controllern bzw. Actions darstellen. Der gewählte Controller greift zur weiteren Verarbeitung auf das Request-Objekt zurück, das einen objektorientierten Zugriff auf die Request-Parameter ermöglicht und über das MvcEvent-Objekt zugänglich gemacht wird. Letzteres wiederum wurde zu Beginn erzeugt und wird an den entsprechenden Stellen verfügbar gemacht. Der Controller bedient sich im Weiteren des Models der Anwendung und greift damit auf persistente Daten, Services und die Businesslogik zu. Er erzeugt schließlich das "View Model", auf dessen Basis und passender HTML-Templates sowie ggf. unter Einsatz sog. "View Helper" das Ergebnis des Aufrufs generiert wird. Optional wird der Output jetzt noch in ein Layout eingefügt und das finale Resultat an den Aufrufer zurückgegeben. Doch mehr zu den Details später.

Da Zend\Mvc umfassend Gebrauch vom Event-Systems des Frameworks macht, bieten sich hier für den Anwendungsentwickler eine ganze Reihe von Möglichkeiten, den oben skizzierten Standardablauf zu beeinflussen. So kann etwa vor der Ausführung des Controllers eine Zugangskontrolle oder ein Eingabefilter realisiert werden. Vor Rückgabe des Ergebnisses könnte so etwa - falls notwendig - mit Hilfe von [HTMLPurifier](http://htmlpurifier.org/)¹⁴ sichergestellt werden, dass das erzeugte HTML-Markup einwandfrei ist.

Der Code der MVC-Implementierung wurde vollständig neu entwickelt. War die MVC-Implementierung in Version 1 des Frameworks noch recht starr, so ist die neue Version deutlich flexibler und erlaubt letztlich die Konfiguration beliebig angepasster Workflows. Im Grunde kann der oben skizzierte Ablauf auf Basis von Zend\Mvc auch vollkommen anders gestaltet werden, ohne dabei auf die Nutzung des Zend Frameworks und der sich daraus ergebenden Vorteile zu verzichten.

3.5 Weitere Komponenten

Neben Zend\Mvc, Zend\View, Zend\ServiceManager, Zend\EventManager und Zend\ModuleManager, die zusammen den "Kern des Frameworks" ausmachen, bringt das Zend Framework 2 noch eine ganze Reihe weiterer Komponenten mit, die wir uns im Folgenden kurz und knapp ansehen wollen. Die meisten Komponenten begegnen uns im Verlauf dieses Buch auch noch einmal im Detail. Wir werden uns dann jeweils intensiver mit der jeweiligen Komponente auseinandersetzen. Im Gegensatz zu vielen anderen Frameworks ist das Zend Framework seit jeher so konzipiert, dass

¹⁴<http://htmlpurifier.org/>

auch nur ausgewählte Komponenten zum Einsatz kommen können, während andere Bestandteile außen vor bleiben.

Beim Blick auf die Liste der Komponenten werden Anwender des Zend Framework 1 feststellen, dass einige Komponenten nicht mehr dabei sind. Insbesondere die vielen Komponenten zur Anbindung diverser Webservices sind mit der Version 2.0 nicht mehr Teil des Frameworks, sondern werden als eigenständige Projekte / Bibliotheken gepflegt. So ist etwa `Zend_Service_Twitter` nicht mehr enthalten, sondern wird nun im Git-Account [zendframework](https://github.com/zendframework)¹⁵ in einem eigenständigen Repository verwaltet. Die Idee dahinter ist die Schärfung des Profils des Frameworks und die Fokussierung seines Einsatzbereiches. Auch gibt es `Zend_Registry` nicht mehr. Dessen Aufgabe wird nun vom `ServiceManager` wahrgenommen, der, wie wir im weiteren Verlauf des Buchs sehen werden, auch deutlich mehr zu leisten vermag. Ebenfalls entfernt wurde `Zend_Test`. Die gute Nachricht hierbei ist, dass aufgrund der im Zend Framework 2 vorherrschenden losen Kopplung der einzelnen Objekte und Services nun im Grunde keine zusätzlichen Funktionen mehr für das Unit Testing erforderlich sind, die nicht bereits über [PHPUnit](https://github.com/sebastianbergmann/phpunit/)¹⁶ selbst gegeben wären. Das sind sehr gute Nachrichten und wie wir in einem späteren Kapitel und im Praxisteil des Buches noch sehen werden, ist das Unit Testing mit der neuen Version grundsätzlich viel einfacher geworden.

Die folgenden Komponenten sind zusätzlich standardmäßig im Zend Framework 2 enthalten:

- **Authentication:** Dient der Implementierung einer "Login"-Funktion, bei der überprüft wird, ob ein Nutzer auch wirklich der ist, für den er sich ausgibt (etwa weil er das geheime Passwort kennt).
- **Barcode:** Bibliothek für das Erzeugen von Barcodes.
- **Cache & Memory:** Generische Implementierung eines Caching-Systems unter Berücksichtigung verschiedener "Backends", etwa "Memcached" oder "APC".
- **Captcha:** Erzeugung von [CAPTCHAs](http://de.wikipedia.org/wiki/CAPTCHA)¹⁷, etwa für die Verwendung in Webforms.
- **Code:** Tools zur automatischen Erzeugung von Code.
- **Config:** Helferlein rund um das Lesen und Schreiben von Applikationskonfigurationen in den unterschiedlichsten Formaten, etwa YAML oder XML.
- **Console:** Bibliothek zur Nutzung von Anwendungsfunktionalität, etwa Controller, über eine Shell (statt auf Basis eines HTTP-Requests eines Browsers).
- **Crypt:** Funktionen rund um "Verschlüsselung".
- **Db:** Bibliothek für die Arbeit mit Datenbanken (allerdings kein ORM-System).
- **Di:** Implementierung eines Dependency Injection (DI)-Containers.
- **Dom:** Bibliothek für die (serverseitige) Arbeit mit dem DOM.
- **Escaper:** Helferlein für das Output-Escaping.
- **Feed:** Erzeugung von RSS- und Atom-Feeds.
- **File:** Helferlein rund um die Arbeit mit Dateien.

¹⁵<https://github.com/zendframework>

¹⁶<https://github.com/sebastianbergmann/phpunit/>

¹⁷<http://de.wikipedia.org/wiki/CAPTCHA>

- `Filter`: Funktionen für das Filtern von Daten, etwa im Rahmen von Webforms.
- `Form`: Bibliothek für die PHP-gestützte, objektorientierte Erzeugung von Webforms unter Berücksichtigung von “Validatoren” und “Filtern”.
- `Http`: Helferlein rund um das HTTP-Protokoll.
- `I18n`: Umfassende Bibliothek für die Internationalisierung von Anwendungen, u.a. die Ausgabe von übersetzten Inhalten.
- `InputFilter`: Ermöglicht die Anwendung von Filtern und Validatoren auf empfangene Daten.
- `Json`: Tools für das Serialisieren und Deserialisieren von JSON-Datenstrukturen.
- `Ldap`: Bibliothek zur Anbindung von LDAP-Systemen, etwa im Zusammenspiel mit `Authentication`.
- `Loader`: Autoloading-Funktionen für PHP-Klassen sowie für das Laden von MVC-Modulen.
- `Log`: Implementierung einer generischen Logging-Funktionalität mit Unterstützung verschiedenartiger “Log-Speicher”.
- `Mail & Mime`: Bibliothek für das Versenden von (Multipart-)E-Mails.
- `Math`: Diverse mathematische Helferlein.
- `Navigation`: Erzeugung und Ausgaben von Website-Navigationen.
- `Paginator`: Erzeugung und Ausgabe von “Blätter-Navigationen”, etwa in Ergebnislisten.
- `Permissions`: Bibliothek für Rechte- und Rollensysteme.
- `ProgressBar`: Erzeugung und Darstellung von Fortschrittsbalken (unter anderem auch auf der Kommandozeile)
- `Serializer`: Tools für das Serialisieren und Deserialisieren von Objekten, etwa zur persistenten Speicherung.
- `Server, Soap & XmlRpc`: Bibliothek für die Erstellung von Webservices, u.a. auf Basis von SOAP oder XML-RPC. Teil von `Soap` ist auch eine hilfreiche SOAP-Client-Implementierung.
- `Session`: Verwaltung von Nutzersessions.
- `Stdlib`: Diverse Standardfunktionen und -objekte, etwa die Implementierung einer “Userland-PriorityQueue”, die unter anderem vom `EventManager` verwendet wird.
- `Tag`: Funktionen zur Verwaltung von “Tags” und der Erzeugung etwa von “Tag-Wolken” auf einer Website.
- `Text`: Helferlein rund um das Management von Strings und Schriften. Wird intern z.B. verwendet von `Captcha`.
- `Uri`: Funktionen für das Erzeugen und Validieren von URIs.
- `Validator`: Umfassende Bibliothek für die syntaktische Validierung von Daten, etwa Formulareingaben, für viele Anwendungsfälle, darunter ISBN, IBAN, E-Mail-Adressen und vieles mehr.
- `Version`: Hält Informationen bereit zur eingesetzten Framework-Version sowie zur bei Github verfügbaren, aktuellsten Version.

3.6 Design-Patterns: Interface, Factory, Manager & Co.

Schaut man eine Weile durch den Zend Framework 2 Code, so stellt man fest, dass er voll gestopft ist mit Implementierungen sogenannter [Design Patterns](#)¹⁸. Ist man eher "typischen PHP-Code" gewohnt - und das ist jetzt in keiner Form wertend gemeint - braucht es eine Weile, bis man sich im Zend Framework 2 zurechtfindet. Hat man es schon einmal mit Java zu tun gehabt, fühlt man sich deutlich schneller zu Hause, einfach weil "Design Patterns" schon einige Jahre früher Einzug in die Java-Welt gehalten, bzw. sich dort sogar herausgebildet haben. Um den Einstieg zu vereinfachen, schauen wir uns im Folgenden ein paar übliche Konstrukte im Zend Framework an. Nicht alle sind tatsächlich Design-Patterns im engeren Sinne, aber über diese Tatsache sehen wir einmal hinweg. Auch darüber, dass ich an manchen Stellen dem Buch dienliche Vereinfachungen in den Erklärungen vornehme. Dies ist kein umfassendes Buch zu Design Patterns und es soll hier vor allem jenes Wissen vermitteln, das hilfreich für das Verständnis des Zend Frameworks ist. An dieser Stelle vielleicht noch einmal folgender Hinweis: Natürlich muss man als Anwendungsentwickler nicht alle Mechaniken des Zend Framework 2 im Detail verstehen. Ganz im Gegenteil: Das Framework soll ja Arbeit abnehmen und es ermöglichen, dass man sich voll und ganz auf die Programmierung der eigentlichen "Business-Logik" fokussieren kann. Dennoch hilft es ungemein, wenn man über ein grundsätzliches Verständnis über die Zusammenhänge der einzelnen Framework-Komponenten - und viel wichtiger noch: der grundlegenden Konzepte - verfügt. Es lohnt sich also, dieses Kapitel nicht zu überspringen.

Interface

Interfaces sind ein inhärenter Bestandteil der objektorientierten Programmierung, die PHP seit Version 5 umfassend unterstützt. Interfaces ermöglichen es, aufrufenden Code von einer konkreten Implementierung zu entkoppeln. Entwickelt man konsequent immer gegen eine definierte Schnittstelle, die garantiert, dass zur Laufzeit tatsächlich eine konkrete Implementierung zur Verfügung steht, die über die vereinbarten Methoden und Eigenschaften verfügt, so kann man jederzeit auch eine alternative Implementierung verwenden, ohne den aufrufenden Code modifizieren zu müssen.

Listener

Ein "Listener" ist ein Stückchen Code, dass ausgeführt wird, sobald in einer Anwendung ein definiertes Ereignis eingetreten ist. Technisch gesprochen wird dazu ein "Listener" zuvor bei einem sog. "EventManager" (Achtung: Verwechslungsgefahr mit einem beliebten Berufszweig!) für ein Ereignis registriert. So kommt die Verbindung an dieser Stelle zusammen.

¹⁸<http://de.wikipedia.org/wiki/Entwurfsmuster>

ListenerAggregate

Ein “ListenerAggregate” hütet eine Reihe von “Listnern” zusammen, um sie etwa gleichzeitig bei einem “EventManager” zu registrieren. Nicht viel mehr, aber auch nicht weniger.

Factory

Eine “Factory” kommt insbesondere immer dann zum Einsatz, wenn die Instanzierung eines bestimmten Objektes aufwändiger ist, also eine ganze Reihe von Handgriffen erforderlich sind, um ein Objekt einsatzbereit zu machen. Etwa nutzt das Zend Framework eine “Factory”, um seinen `ModuleManager` zu instanzieren und zusätzlich einige modulrelevante “Listener” zu registrieren.

Service

Ein “Service” stellt den Zugriff auf bestimmte Daten oder Funktionen zur Verfügung. Der Begriff ist sehr breit gefasst und kann je nach Kontext eine vollkommen andere Bedeutung haben. Im Rahmen des Frameworks versteht man unter einem “Service” ein Objekt, dass von einem `ServiceManager` bereitgestellt wird und einen definierten Dienst anbietet. So werden etwa im `ServiceManager` registrierte “Listener” als “Service” bezeichnet, ebenso etwa der `ModuleManager`, aber auch Controller oder “View Helper”.

Manager

Ein “Manager” ist ein Objekt, dass sich um die Verwaltung eines bestimmten Typs anderer Objekte in einem System kümmert. [Doctrine 2](http://www.doctrine-project.org/)¹⁹ etwa hat einen sogenannten “EntityManager”, der Entities, also gewisse persistente Objekte (in einem Shop z.B. Angebote, Kategorien, Kunden, Bestellungen, usw.) über deren Lebenszyklus hinweg verwaltet und dafür sorgt, dass Entitäten aus einer Datenbank gelesen und Änderungen transparent zurückgeschrieben werden.

Strategy

Hinter dem Strategy-Design-Pattern steht die Idee, Algorithmen, die man andernfalls an der jeweiligen Stelle im Code “hart verdrahten” würde, in eine eigene Klasse auszulagern und so einfach austauschbar zu machen. So sind Sortieralgorithmen etwa ein guter Kandidat für das Strategy-Pattern: Die unterschiedlichen Algorithmen, nach denen z.B. eine Produkt-Liste sortiert werden kann (Preis auf-/absteigend, Bewertung auf-/absteigend, usw.) werden nicht fest einkodiert, sondern jeweils in Form eigener Klassen realisiert, die alle ein gemeinsames Interface implementieren, das eine Methode `sort()` vorgibt. Hat man diesen Mechanismus einmal implementiert, kann zu einem späteren Zeitpunkt ein beliebiges, anderes Sortierverfahren realisiert und “eingehängt” werden.

¹⁹<http://www.doctrine-project.org/>

Model-View-Controller (MVC)

Das MVC-Muster beeinflusst maßgeblich die Struktur des Anwendungscode, weil es logisch die Bestandteile voneinander trennt, die sich um die Darstellung (View), die Verarbeitung der Nutzerinteraktion (Controller) und die “Business-Logik” mit ihren Objekten und Services kümmern.

Actions

“Actions” sind ein Ansatz, um den Code zur Verarbeitung von Nutzerinteraktionen in Controllern weiter zu strukturieren. Sie sind daher eng mit dem MVC-Muster verbunden und kommen auch im Zend Framework 2 zum Einsatz.

View Helper

Mit Hilfe von “View Helfern” lässt sich Code für die Darstellungslogik kapseln und auf standardisierte Art und Weise wiederverwenden.

Controller Plugins

Über “Controller Plugins” lässt sich häufig genutzter Code für die Interaktionsverarbeitung organisieren und in mehreren Controllern verwenden.

4 Hallo, Zend Framework 2!

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

Wie im letzten Kapitel besprochen, handelt es sich bei Zend\Mvc um eine eigenständige, optionale, aber maßgebliche Komponente des Frameworks. Die folgenden Inhalte beziehen Zend\Mvc immer mit ein. Zend\Mvc diktiert der eigenen Anwendung auch in gewisser Weise die Verzeichnis- und Codestruktur. 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.



Zend Framework 1 und 2 parallel

Man kann das Zend Framework 2 problemlos auch neben Version 1 betreiben und sich beim Zend Framework 2 zunächst nur punktuell bedienen.

4.1 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 eine Zend\Mvc-basierte Anwendung tatsächlich in Aktion zu sehen. 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”

¹<http://packages.zendframework.com/>

erstellt, die als Vorlage für das eigene Projekt dient und den notwendigen “Boilerplate-Code” mitbringt, den man sonst aufwändig selbst erstellen müsste.

4.2 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! Es ist kein weitergehendes Git-Wissen erforderlich, um mit diesem Buch und dem Framework erfolgreich zu arbeiten. Es steht dem Leser natürlich frei, im Weiteren den eigenen Anwendungscode auch dauerhaft mit Git zu verwalten, 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/>

4.3 Composer

Die “ZendSkeletonApplication” macht sich mit [Composer](https://getcomposer.org/)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": "2.*"
13    }
14 }
```

Listing 4.1

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 der aktuellen Version benötigt. Die folgenden zwei 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 gemacht.

³getcomposer.org/



Phar-Archiv

Ein Phar-Archiv bietet die Möglichkeit, eine PHP-Anwendung in Form einer einzelnen Datei zur Verfügung zu stellen. Wirft man einen Blick auf das [Composer-Repository bei github](#)⁴ wird deutlich, dass Composer nicht, wie man zunächst denken könnte, nur aus einer einzigen Datei bestünde, sondern dessen Bestandteile lediglich zur Verbreitung der Anwendung in ein Phar-Archiv gebündelt wurden.



Phar-Archive und Suhosin

Kommt “[Suhosin](#)”⁵ auf einem System zum Einsatz, muss die Nutzung von Phar zunächst explizit erlaubt werden, indem die `suhosin.ini` um den Eintrag `suhosin.executor.include.whitelist=phar` erweitert wird. Andernfalls kann es bei der Ausführung von Composer-Kommandos zu Problemen kommen.



Installation ohne Git oder Composer

Es ist bei Bedarf ebenso möglich, das Framework und die “`ZendSkeletonApplication`” über einen “normalen Download” oder Pyrus (den PEAR-Nachfolger) zu beziehen. Weitere Installationshinweise finden sich auf der [offiziellen Download-Site](#)⁶.

4.4 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 Webserver 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

⁴<https://github.com/composer/composer>

⁵<http://www.hardened-php.net/suhosin/>

⁶<http://framework.zend.com/downloads>

```
1 [..]  
2 DocumentRoot /var/www/ZendSkeletonApplication/public  
3 [..]
```

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

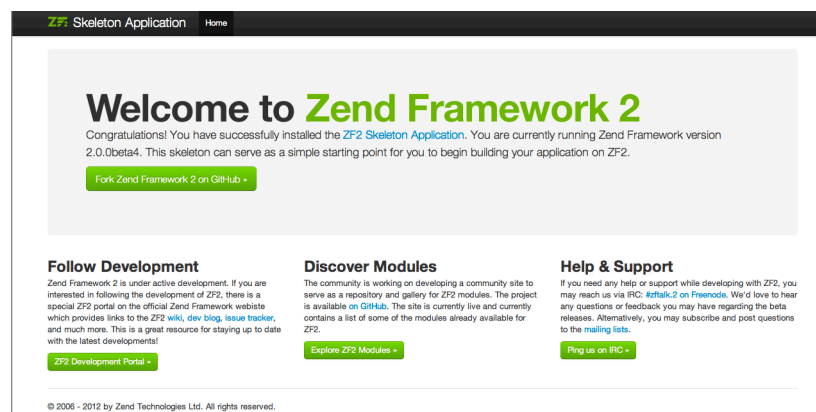
```
1 $ cd /var/www  
2 $ git clone git://github.com/zendframework/ZendSkeletonApplication.git
```



Einrichten einer PHP-Laufzeitumgebung

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

4.5 Verzeichnisstruktur einer Zend Framework 2 Anwendung

Jetzt haben wir sie also vor uns, eine Zend\Mvc-basierte Zend Framework 2 Anwendung, mit ihrem charakteristischen Verzeichnislayout und dem typischen Konfigurations- und Initialisierungscode:

```
1  ZendSkeletonApplication/
2      config/
3          application.config.php
4      autoload/
5          global.php
6          local.php
7      ...
8      module/
9      vendor/
10     public/
11         .htaccess
12         index.php
13     data/
```

Das “Application Root” ist in unserem Fall das Verzeichnis `ZendSkeletonApplication`, das automatisch beim Klonen des entsprechenden github-Repository 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 wird insbesondere der `ModuleManager` konfiguriert, auf dessen 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 weist darauf hin, dass die Konfigurationen, die in diesem Verzeichnis abgelegt sind, automatisch berücksichtigt 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 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 “hartverdrahten”, 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. So lassen sich auch spezielle Konfigurationen für Test- oder Staging-Systeme hinterlegen. Übrigens werden auch Konfigurationsdateien der Form “xyz.local.php” (gilt ebenso für “global”) , etwa db.local.php wie beschrieben vom Framework berücksichtigt.

Im Verzeichnis `module` befinden sich die einzelnen Module der Anwendung. Jedes Modul selbst kommt mit einem eigenen, typischen Verzeichnisbaum daher, den 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` (bzw. dessen “Spezialisierungen” wie oben beschrieben), 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 wichtig sein und dankenswerter Weise durch den `ServiceManager` greifbar gemacht. Auch darüber werden wir später noch mehr erfahren. Der aufmerksame Leser stellt an dieser Stelle fest, dass durch diese “Konfigurations-Kaskade” 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 den Code, den man nicht selbst geschrieben hat (vom Code der “`ZendSkeletonApplication`” an dieser Stelle einmal abgesehen, aber den hätte man im Zweifel ja auch selbst schreiben müssen), bzw. den man nicht speziell für diese eine Anwendung geschrieben hat. Das Zend Framework 2 befindet sich also etwa dort, aber ggf. auch weitere Bibliotheken. 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 zusätzlicher Bibliotheken sollte daher im Idealfall also immer über `Composer` durchgeführt werden. Interessant ist auch die Tatsache, dass auch ZF2-Module, die ja eigentlich im Verzeichnis `module` ihren Platz haben, auch über das Verzeichnis `vendor` bereitgestellt werden können (korrekterweise muss man sagen, dass dies grundsätzlich über die `application.config.php` konfigurierbar ist und Module daher im Grunde überall abgelegt werden können). Das heißt, dass sich auch Bibliotheken 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 im Rahmen der jeweiligen Anwendung entwickelt hat. Alle weiteren 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 für den “zentralen Einstiegspunkt”, die `index.php`. Die Idee dabei ist: 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 eingelesen 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](#)⁷ 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.).

4.6 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 erinnern 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:

⁷http://httpd.apache.org/docs/current/mod/mod_rewrite.html


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

Listing 4.1

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 // [...]
3 if (file_exists('vendor/autoload.php')) {
4     $loader = include 'vendor/autoload.php';
5 }
6 // [...]
```

Listing 4.2

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 mehr einfacher sein, Bibliotheken in die eigene Anwendung zu integrieren!

In der `init_autoloader.php` wird dann alternativ noch das Autoloading der ZF2-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 genannte 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 // [...]
3 Zend\Mvc\Application::init(
4     include 'config/application.config.php')
5     ->run();
```

Listing 4.3

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 Einstiegspunkt 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` sinngemäß als “Services” bezeichnet - 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 Instanziierung 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 zum Code: Der Methode `init()` wird als Parameter die Anwendungskonfiguration übergeben und diese bereits bei der Erzeugung des `ServiceManager` berücksichtigt:

```
1  <?php
2  // [...]
3  $serviceManager = new ServiceManager(
4      new ServiceManagerConfig(
5          $configuration['service_manager']
6      )
7  );
8  // [...]
```

Listing 4.4

Hier wird nun der `ServiceManager` initialisiert und mit den Services ausgestattet, die im Rahmen der Verarbeitung von Requests durch `Zend\Mvc` erforderlich sind. Der `ServiceManager` kann allerdings grundsätzlich auch für vollkommen andere Zwecke genutzt werden kann, jenseits von `Zend\Mvc`.

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

```
1  <?php
2  // [...]
3  $serviceManager->setService('ApplicationConfig', $configuration);
4  // [...]
```

Listing 4.5

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

```
1 <?php
2 // [...]
3 $serviceManager->get('ModuleManager')->loadModules();
4 // [...]
```

Listing 4.6

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

```
1 <?php
2 // [...]
3 $serviceManager = new ServiceManager(
4     new ServiceManagerConfig($configuration['service_manager'])
5 );
6 // [...]
```

Listing 4.7

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 aber 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 über die `application.config.php` aktivierten Module. Sind die Module bereit, wird der `ServiceManager` einmal mehr gefordert und um den Service “`Application`” gebeten:

```
1 <?php
2 // [...]
3 return $serviceManager->get('Application')->bootstrap();
4 // [...]
```

Listing 4.8

Diese Tatsache mutet etwas merkwürdig an, hat die gesamte Verarbeitung doch ursprünglich bereits über `Zend\Mvc\Application` ihren Anfang genommen. Doch jetzt wird klar, dass dessen `init()`-Methode zunächst nur den `ServiceManager` initialisiert, die eigentliche `Application` dann aber selbst als `Service` erzeugt wird.

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:

```
1 <?php
2 // [...]
3 Zend\Mvc\Application::init(include 'config/application.config.php')
4     ->run();
5 // [...]
```

Listing 4.9

Der genauen Betrachtung der eigentlichen Request-Verarbeitung habe ich aufgrund seiner Wichtigkeit, aber auch Komplexität, ein eigenes Kapitel in diesem Buch gewidmet. Wir halten bis hierhin 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 einen passenden Controller samt Action ausfindig zu machen. Der `ServiceManager` steht im Zentrum der Verarbeitung und gibt Zugriff auf die `Services` der Anwendung. Daher müssen wir zunächst den `ServiceManager` erzeugen, bevor dieser uns wiederum Zugriff auf den `ModuleManager` gibt, mit dessen Hilfe wir die registrierten Module zum Leben erwecken, als auch auf die `Application`, die sich für die Verarbeitung der Requests verantwortlich fühlt. So weit, so gut.



Zend Framework 2 mit alternativen Webservern

Selbstverständlich kann anstelle des Apache auch ein alternativer Webserver wie [nginx](http://nginx.org/)⁸ zum Einsatz kommen. In diesem Fall sind lediglich die Apache-spezifische Konfiguration, wie die der `.htaccess` analog umzusetzen, etwa mit Hilfe der “nginx rules”.

⁸<http://nginx.org/>

5 Ein eigenes Modul erstellen

Die eigentliche Anwendungslogik, also die einzelnen Seiten, Templates, Formular usw. wird in Modulen gekapselt. Jetzt, wo wir die lauffähige ZendSkeletonApplication verfügbar haben, ist es an der Zeit, das erste eigene Modul zu erstellen. Weil wir uns zunächst auf die Handgriffe konzentrieren wollen, die notwendig sind, um ein eigenes Modul zu erstellen und einzubinden, starten wir mit dem Klassiker: Hello, World!

5.1 Das “Hello World” - Modul erstellen

Ein Zend Framework 2 - Modul ist zunächst einmal charakterisiert durch eine definierte Verzeichnisstruktur und einige Dateien, die es in jedem Modul geben muss oder die es bei Bedarf geben kann:

```
1  Module.php
2  config/
3      module.config.php
4  public/
5      images/
6      css/
7      js/
8  src/
9      Helloworld/
10         Controller/
11             IndexController.php
12  view/
13      helloworld/
14          index/
15              index.phtml
```

Diese Struktur muss in einem Verzeichnis `Helloworld` im Verzeichnis `module` innerhalb der Anwendung angelegt werden. Per Konvention stellt ein Modul einen eigenen Namensraum dar, den wir dank PHP 5.3 jetzt auch nativ so auszeichnen können. Bei Version 1 des Frameworks mussten noch die Pseudo-Namensräume Anwendung finden, die zu sehr langen Klassenbezeichnungen, etwa zu `Zend_Form_Decorator_Captcha_Word`, geführt haben. Dieses Problem gehört seit PHP 5.3 und dem Zend Framework 2 zum Glück der Vergangenheit an.

Zunächst füllen wir die Datei `Module.php` mit Leben:

```
1  <?php
2  namespace Helloworld;
3
4  class Module
5  {
6      public function getConfig()
7      {
8          return include __DIR__ . '/config/module.config.php';
9      }
10 }
```

Listing 5.1

Die Klasse `Module` ist dem Namensraum zugeordnet, der uns durch unser Modul vorgegeben wird, in diesem Fall `Helloworld`. Bei der Klasse selbst handelt es sich um eine normale PHP-Klasse, die über eine Reihe von Methoden verfügen kann, die durch die unterschiedlichen Manager und Komponenten des Frameworks etwa im Rahmen der Initialisierung aufgerufen werden. Dazu gehört auch die Methode `getConfig()`. Wie schon im Zend Framework 1, findet auch in der neuen Version der Ansatz “Convention over configuration” regen Einsatz. Meint, es gibt bestimmte Übereinkünfte, die, wenn so angewandt wie vereinbart, eine weitere Konfiguration überflüssig machen. In diesem Fall wurde die folgende Konvention festgeschrieben: Wenn du in deiner `Module`-Klasse eine Methode `getConfig()` implementierst, wird sie im Rahmen der Initialisierung vom `ModuleManager` aufgerufen. Gesagt, getan. Unsere Methode liefert aber nicht gleich selbst die Konfiguration zurück, sondern liest dazu das File `module.config.php` im Verzeichnis `config` des Moduls ein, das folgenden Inhalt bekommt:

```
1  <?php
2  return array(
3      'view_manager' => array(
4          'template_path_stack' => array(
5              __DIR__ . '/../view'
6          )
7      )
8  );
```

Listing 5.2

Zunächst einmal fällt auf, dass die Konfiguration für ein Modul über ein PHP-Array abgebildet wird. Es gibt ja grundsätzlich eine ganze Reihe von Möglichkeiten, wie man Konfigurationen vorhalten kann, etwa als INI-Datei, via YAML oder als XML-Struktur. All diese Strukturen erfordern ein mehr oder weniger aufwändiges Parsing. Die effizienteste und im Framework favorisierte Methode ist es jedoch, die Konfiguration gleich in PHP-Code abzulegen. Dies macht jegliches Parsing unnötig und ein schlankes `include()` sorgt bereits für den gewünschten Effekt des Einlesens der

Konfiguration. Auch hier schlägt “Convention over configuration” wieder zu: Wenn wir in unserer Konfigurationen einen Abschnitt `view_manager` definieren, werden diese Werte später auf magische Art und Weise bei der Suche nach dem richtigen Template berücksichtigt. Wir konfigurieren hier also, in welchem Verzeichnis die Views (die HTML-Templates) unseres Moduls abgelegt sind. An dieser Stelle demnach also kein “Convention over configuration”, sondern eine explizite Angabe.

Weiterhin sollten wir in der `Module.php` angeben, wie das Autoloading der einzelnen Klassen des Moduls funktioniert. Dazu implementieren wir die Methode `getAutoloaderConfig()`, die vom `ModuleManager` wieder per Konvention bei der Initialisierung des Moduls verarbeitet wird:

```

1  <?php
2  // [...]
3  public function getAutoloaderConfig()
4  {
5      return array(
6          'Zend\Loader\StandardAutoloader' => array(
7              'namespaces' => array(
8                  __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__
9              )
10         )
11     );
12 }
13 // [...]
```

Listing 5.2

Wir werfen etwas später noch einmal einen genaueren Blick auf das Thema “Autoloading” und begnügen uns an dieser Stelle mit dem Wissen, dass wir mit dem oben aufgeführten Konstrukt dafür sorgen, dass die Klassen dieses Modules - insbesondere auch die Controller - automatisch geladen und damit auch vom Framework berücksichtigt werden können. Die Datei `Module.php` sieht jetzt also wie folgt aus:

```

1  <?php
2  namespace HelloWorld;
3
4  class Module
5  {
6      public function getAutoloaderConfig()
7      {
8          return array(
9              'Zend\Loader\StandardAutoloader' => array(
10                 'namespaces' => array(
11                     __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__
```

```
12         )
13     )
14 );
15 }
16
17     public function getConfig()
18     {
19         return include __DIR__ . '/config/module.config.php';
20     }
21 }
```

Listing 5.3

Nun widmen wir uns dem IndexController im Verzeichnis `/src/Helloworld/Controller`:

```
1  <?php
2  namespace Helloworld\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6
7  class IndexController extends AbstractActionController
8  {
9      public function indexAction()
10     {
11         return new ViewModel(array('greeting' => 'hello, world!'));
12     }
13 }
```

Listing 5.4

Zunächst einmal achten wir wieder auf den Namespace. Unser IndexController gehört also zum Modul `Helloworld` und ist dort ein Controller. Soweit, so logisch. Die Klasse erbt von der Framework-Klasse `Zend\Mvc\Controller\AbstractActionController`, was sie zu dem macht, was sie ist: Eine Klasse, die einen Request verarbeiten kann (“dispatching”) und sich dabei seiner “Actions” bedient. “Actions” meint öffentliche Methoden, die wieder einer bestimmten Namenskonvention folgen: Eine Methode der Klasse wird dann zu einer “Action”, wenn der Methodenbezeichnung ein “Action” angehängt wird. Der Controller verfügt derzeit also über eine Action, die `indexAction`.

Innerhalb der `indexAction` passieren jetzt üblicherweise eine ganze Menge an Dingen, wie etwa die Verarbeitung von Request-Parametern, das Schreiben oder Lesen von Daten aus Datenbanken oder der Zugriff auf entfernte Webservices. Üblicherweise macht die Action das nicht alles selbst (man spricht in diesem Zusammenhang ansonsten auch vom sog. “Fat Controller”), sondern delegiert

einzelne Aufgaben an andere Mitstreiter. Dieses Vorgehen erhöht meist die Wiederverwendung und Wartbarkeit des Codes.

In aller Regel beendet eine Action ihre Arbeit damit, dass sie die Ergebnisse der durchgeführten Operationen für die Darstellung im Browser des Aufrufers bereitstellt. Im Zend Framework 2 retourniert (sagt man sonst nur im Tennis!) alle notwendigen Daten in Form eines sog. “View Models”. Einfach gesagt repräsentiert ein “View Model” die hinter einem “User Interface (UI)” stehenden Daten und steuert zudem auch den Status bestimmter UI-Komponenten. Wir gehen da später noch einmal detaillierter drauf ein.

Wenn wir die Grußformel “hello, world!” jetzt als Überschrift in einem Browser anzeigen wollen, fehlt uns noch der entsprechende `h1` - Tag. Da sich um die HTML-Darstellung in einer MVC-basierten Anwendung die sog. “View” kümmert, müssen wir genau diese jetzt noch erstellen (korrekterweise erstellen wir eigentlich nur ein “View-Template”, das im Rahmen des “Rendering” einer “View” und auf Basis eines “View Models” das gewünschte Ergebnis erzeugt). Zu jeder Action gibt es in der Regel genau eine View, bzw. ein View-Template. Dazu tragen wir folgendes in die Datei `index.phtml` im Verzeichnis `view/helloworld/index` ein:

```
1 <h1><?php echo $this->greeting; ?></h1>
```

Listing 5.5

Auch in der Strukturierung der Views spielt wieder der Namensraum des Moduls, aber auch die Bezeichnung des Controllers sowie die der Action eine Rolle, wie sich gut erkennen lässt. Unsere View liegt in einem Verzeichnis `view`. Das ist fix und ändert sich nicht. Dann in einem Unterverzeichnis, das nach dem Modul benannt ist und dort nochmals in einem Unterverzeichnis, das nach dem Controller benannt ist. Die View selbst trägt den Namen der jeweiligen Action samt dem Suffix “.phtml”. Eine Datei mit der Endung “.phtml” besteht per Konvention aus PHP-Code und HTML-Markup, wobei sich der PHP-Code auf die Darstellung beschränken und nicht etwa Business-Logik enthalten sollte. Der Suffix “phtml” steht übrigens für PHP + HTML. In unserem View-File haben wir also HTML-Markup und greifen dann via `$this` auf das Datenmodell der View zu, welches wir zuvor im Controller erzeugt haben. Mit “greeting” greifen wir konkret auf den Key `greeting` zu, für den wir im View-Model den Wert “hello, world!” hinterlegt haben. Den geben wir dann via `echo` aus.

Aber noch sind wir nicht fertig. Wir wollen, dass wir “hello, world!” auf dem Bildschirm sehen, wenn wir die URL `http://localhost/sayhello` aufrufen. Dazu müssen wir die Konfiguration des Moduls in der `module.config.php` noch um eine entsprechende Route und die Details zu unserem Controller erweitern:

```
1  <?php
2  return array(
3      'view_manager' => array(
4          'template_path_stack' => array(
5              __DIR__ . '/../view',
6          ),
7      ),
8      'router' => array(
9          'routes' => array(
10             'sayhello' => array(
11                 'type' => 'Zend\Mvc\Router\Http\Literal',
12                 'options' => array(
13                     'route' => '/sayhello',
14                     'defaults' => array(
15                         'controller' => 'Helloworld\Controller\Index',
16                         'action' => 'index',
17                     )
18                 )
19             )
20         )
21     ),
22     'controllers' => array(
23         'invokables' => array(
24             'Helloworld\Controller\Index'
25                 => 'Helloworld\Controller\IndexController'
26         )
27     )
28 );
```

Listing 5.6

Analog zum Key `view_manager` sorgt der Key `router` per Konvention dafür, dass die Konfiguration des Routings der entsprechenden Framework-Komponente zugänglich gemacht werden. Da wir uns mit dem Routing später noch detaillierter beschäftigen werden, sei an dieser Stelle nur so viel gesagt: Wir übergeben hier dem Router ein Array mit einzelnen Routen, von denen wir eine mit “sayhello” benannt haben, die immer dann greifen soll, wenn in der URL nach der Hostinformation (in unserem Fall localhost) der String “/sayhello” folgt. Ist dies der Fall, soll das Framework dafür sorgen, dass unser gerade erstellter `IndexController` und darin die Action `index` ausgeführt wird. Das ist eigentlich schon alles. Durch die geschachtelte Array-Notation sieht die Konfiguration zunächst etwas unübersichtlich aus. Nach kurzer Zeit hat man sich aber schnell daran gewöhnt.

Auffällig ist, dass wir als Wert für den Controller `Helloworld\Controller\Index` angegeben haben, obwohl der Controller ja `IndexController` heißt. Etwas weiter unten findet sich die Erklärung:

```
1 <?php
2 // [...]
3 'controllers' => array(
4     'invokables' => array(
5         'Helloworld\Controller\Index'
6         => 'Helloworld\Controller\IndexController'
7     )
8 )
9 // [...]
```

Listing 5.7

Mit diesem Stückchen Code definieren wir für unseren Controller einen über alle Module der Anwendung hinweg gültigen, eindeutigen Namen. Um die Eindeutigkeit sicherzustellen, haben wir der Bezeichnung des Controllers den Namen des Moduls vorangestellt: `Helloworld\Controller\Index` ist nun also der symbolische Name für unseren Controller, den wir entsprechend bei der Routenkonfiguration verwenden.

Last but not least müssen wir nun die Datei `application.config.php` noch dahingehend erweitern, dass unser neues Modul überhaupt berücksichtigt wird. Dazu ergänzen wir im Abschnitt `modules` den Namen unseres Moduls:

```
1 <?php
2 // [...]
3 'modules' => array(
4     'Application',
5     'Helloworld'
6 )
7 // [...]
```

Listing 5.8

Die URL `http://localhost/sayhello` sollte nun das gewünschte Ergebnis bringen und “hello, world!” auf dem Bildschirm ausgeben.

5.2 Autoloading

Üblicherweise müssen Klassen, bevor sie das erste Mal genutzt werden können, zunächst durch einen `require()` - Aufruf (oder ähnlich) verfügbar gemacht werden, ist es doch so, dass während der Ausführung eines Skripts, dort nur der Code greifbar ist, der dem PHP-Interpreter zuvor auch zugänglich gemacht wurde. Es reicht also nicht aus, eine PHP-Klasse zu programmieren, diese irgendwo auf dem Dateisystem abzulegen und jene in einem anderen, sich in der Ausführung

befindlichen Skript zu referenzieren, ohne, dass man sie zuvor bekannt gemacht hätte. An dieser Stelle ein kurzer Exkurs: Es wird grundsätzlich unterschieden zwischen Code, der Teil des PHP-Kerns ist und dem sog. “Userland” - Code. Während sich z.B. die Core-Klasse `\DateTime` ohne vorherige “Bekanntmachung” nutzen lässt, gilt das für selbstgeschriebene Klassen, also “Userland” - Code, nicht. Er muss zunächst immer dem PHP-Interpreter bekannt gemacht, die jeweiligen PHP-Files, in denen sich die Klassendefinitionen befinden, geladen werden.

Das Zend Framework 2 nutzt sehr intensiv Autoloading. Autoloading meint schlichtweg, dass die Bekanntmachung von Klassen automatisch stattfindet, die jeweiligen PHP-Files mit den dort definierten Klassen also selbstständig bei Bedarf geladen werden. Damit dies funktioniert, muss - wie wir bereits in der Methode `getAutoloaderConfig()` in der Datei `Module.php` gesehen haben - eine gewisse Konfiguration getätigt werden.

Zwar sparen wir uns mit dem Autoloading auf der einen Seite eine Menge Schreibarbeit, weil wir ansonsten jede Datei über ein explizites `require()` laden müssten, belasten das System aber auf der anderen Seite nun zusätzlich mit der Aufgabe des Autoloadings. Autoloading hat also gewisse Kosten, die sich spürbar auf die Ausführungszeit einer Zend Framework 2 Anwendung auswirken können. Das Gute ist: Es gibt unterschiedliche Arten, darunter auch performante, wie Autoloading realisiert werden kann, die allesamt vom Framework unterstützt werden. Das Framework bringt zwei Maßgebliche Klassen mit, die für das Autoloading eingesetzt werden.

Standard-Autoloader

Der Standard-Autoloader ist die Implementierung der mittlerweile üblichen Art und Weise, Autoloading zu realisieren. Dabei wird der Klassenname 1-zu-1 auf einen Dateinamen übersetzt. So erwartet der entsprechende Loader etwa, dass die Klasse `Zend_Translate` (aus dem Zend Framework 1) in einer Datei `Translate.php` im Verzeichnis `Zend` definiert ist. Für Klassen, die Gebrauch von “echten Namespaces” machen, gilt dies ebenso: Eine Klasse `Translate`, die im Namespace `Zend` definiert ist, wird auf dem Filesystem an der gleichen Stelle erwartet. Diese Konvention entspricht sowohl dem [PEAR¹](http://pear.php.net/)-Standard, als auch dem [PSR-0²](https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md) der [PHP Framework Interoperability Group³](https://github.com/php-fig/fig-standards). Wichtig ist, dass man daran denkt, für den jeweiligen (Pseudo-)Namespace anzugeben, wo sich im Dateisystem das entsprechende Verzeichnis befindet, so wie wir dies in der Datei `Module.php` getan haben:

¹<http://pear.php.net/>

²<https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md>

³<https://github.com/php-fig/fig-standards>

```

1  <?php
2  namespace HelloWorld;
3
4  class Module
5  {
6      public function getAutoloaderConfig()
7      {
8          return array(
9              'Zend\Loader\StandardAutoloader' => array(
10                 'namespaces' => array(
11                     __NAMESPACE__
12                         => __DIR__ . '/src/' . __NAMESPACE__,
13                 ),
14             ),
15         );
16     }
17
18     // [...]
19 }

```

Listing 5.9

Der `include_path` wird nämlich nicht mehr herangezogen, um das Laden von Klassen bestmöglich zu beschleunigen. Mehr braucht man hier eigentlich schon gar nicht wissen. Hält man sich an diese Konvention, werden die Klassen problemlos automatisch geladen.

ClassMapAutoloader

Die performanteste Implementierung des Autoloading ist allerdings der `ClassMapAutoloader`, der auf einem simplen, assoziativen PHP-Array operiert, das als Key jeweils den voll qualifizierten Klassennamen und als Wert den passenden Dateinamen enthält, also etwa so aussieht:

```

1  <?php
2  return array(
3      'PhlyContact\Service>ContactControllerFactory'
4          => __DIR__ . '/src/PhlyContact/' .
5              'Service/ContactControllerFactory.php',
6  );

```

Listing 5.10

Wird die entsprechende Klasse angefordert, schaut der Loader in dem Array nach dem passenden Wert und lädt die Datei. Fertig. Der Nachteil hier liegt auf der Hand: Die `ClassMap` will dauerhaft

gepflegt werden. Findet sich hier eine bestimmte Klasse nicht, schlägt das Autoloading fehl. Glücklicherweise gibt es zum einen aber die Möglichkeit, eine ClassMap automatisch generieren zu lassen (z.B. im Rahmen eines Build-Prozesses) um sicherzustellen, dass man keine Klasse vergessen hat und zum anderen können mehrere Arten und Weisen des Autoloadings kombiniert werden:

```
1  <?php
2  public function getAutoloaderConfig()
3  {
4      return array(
5          'Zend\Loader\ClassMapAutoloader' => array(
6              __DIR__ . '/autoload_classmap.php'
7          ),
8          'Zend\Loader\StandardAutoloader' => array(
9              'namespaces' => array(
10                 __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
11             ),
12         ),
13     );
14 }
```

Listing 5.11

Auf diese Art und Weise wird zunächst die ClassMap konsultiert und notfalls auf den PSR-0-Mechanismus zurückgegriffen, wenn es bis dahin keinen Treffer gab.

Die eigentliche ClassMap liegt in einer eigenen Datei jenseits der `Module.php` und wird von dort nur referenziert. Die Idee dahinter ist, das Vorgehen zur Realisierung des Autoloading soweit zu Verallgemeinern, auch jenseits des Zend Frameworks, dass sich Bibliotheken unterschiedlicher Quellen problemlos integrieren lassen. Einige Code-Bibliotheken haben in den letzten Jahren bereits erkannt, dass es einen echten Mehrwert für den Anwender darstellt, wenn die Bibliothek irgendeine Form von Unterstützung mitbringt, die Bibliotheksklassen automatisch zu laden. Das Problem daran ist, dass es sich jeweils wieder um Insellösungen handelt. Jede Bibliothek geht in der konkreten Realisierung des Autoloading im Zweifelsfall einen eigenen, wenn auch ähnlichen, im Detail aber doch unterschiedlichen Weg. Im Rahmen der Module einer ZF2-Anwendung orientiert sich das Framework dazu an einem zusätzlichen Standard, von dem etwa das zuvor angesprochene Composer profitiert und auf diesem Weg bibliotheksübergreifendes Autoloading auf einfachste Art und Weise zu ermöglichen ist. Dazu werden jedem Modul drei zusätzliche, autoloadingrelevante Files spendiert: `autoload_classmap.php`, `autoload_function.php` und `autoload_register.php`.

Die `autoload_classmap.php` gibt ein PHP-Array als Mapping von Klassennamen und Dateinamen wie oben beschrieben zurück, die ein beliebiger Autoloader, nicht nur der des Zend Frameworks, sondern etwa auch der von Composer, verarbeiten und etwa bei Bedarf ggf. auch noch mit ClassMaps anderer Bibliotheken vereinigen kann. `autoload_function.php` hingegen liefert eine PHP-Funktion zurück:

```
1  <?php
2  return function ($class) {
3      static $classmap = null;
4      if ($classmap === null) {
5          $classmap = include __DIR__ . '/autoload_classmap.php';
6      }
7      if (!isset($classmap[$class])) {
8          return false;
9      }
10     return include_once $classmap[$class];
11 };
```

Listing 5.12

Die zurückgelieferte Funktion kann von einem Autoloader ebenfalls verarbeitet werden, etwa in der Form, dass sie als zusätzliche Quelle für das Autoloading von Klassen berücksichtigt wird. Letztlich greift aber auch diese Funktion dann wieder auf die ClassMap zu. Und `autoload_register.php` kommt noch schlanker daher:

```
1  <?php
2  spl_autoload_register(include __DIR__ . '/autoload_function.php');
```

Listing 5.13

Hier wird jetzt also die zuvor definierte Autoloading-Funktion, die wiederum auf die eigentliche ClassMap zugreift, direkt für das Autoloading registriert und gar nicht mehr an den Aufrufer zurückgegeben. Ein simples

```
1  <?php
2  require_once 'autoload_register.php';
```

Listing 5.14

sorgt dann also dafür, dass das Autoloading für ebenjene Komponente funktioniert, allerdings ohne die Möglichkeit, weitere Optimierungen, etwa in der Ausführungsreihenfolge aller registrierten Autoloading-Funktionen, vorzunehmen.

Die drei Autoloading-Files sind, wie wir im vorangegangenen Kapitel bereits gesehen haben, nicht erforderlich, damit ein Modul funktioniert, beziehen sich doch all diese Files auf das ClassMap-Autoloading, das selbst nicht zwingend notwendig ist, aber die Performance einer Anwendung spürbar verbessern kann.

Ein vollständiges Module-Verzeichnislayout für das Helloworld-Modul samt der Autoloading-Files stellt sich also dann abschließend wie folgt dar:

```
1  Module.php
2  autoload_classmap.php
3  autoload_function.php
4  autoload_register.php
5  config/
6      module.config.php
7  public/
8      images/
9      css/
10     js/
11  src/
12      Helloworld/
13          Controller/
14              IndexController.php
15  views/
16      helloworld/
17          index/
18              index.phtml
```


6 Einmal Request und zurück

Schauen wir uns an, was genau mit einem Request passiert und wie die verschiedenen Kollaborateure des Frameworks die Antwort mit unserem “hello, world!” an den Browser generieren.

Alles beginnt mit dem Aufruf der URL `http://localhost/sayhello`. Der HTTP-Request erreicht den Webserver, der nach Konsultation der `.htaccess` entscheidet, dass die `index.php` durch den PHP-Interpreter verarbeitet werden muss. Im Rahmen der `index.php` wird zunächst das Autoloading konfiguriert und im Anschluss die `init()` - Methode der Application aufgerufen:

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

Listing 6.1

6.1 ServiceManager

Dort wird der ServiceManager instanziiert:

```
1 <?php
2 // [...]
3 $serviceManager = new ServiceManager(
4     new ServiceManagerConfig($configuration['service_manager'])
5 );
6 // [...]
```

Listing 6.2

Durch die Übergabe der `ServiceManagerConfig` wird der `ServiceManager` mit einigen Standard-Services ausgestattet, die `Zend\Mvc` für seinen reibungslosen Betrieb benötigt. Der `ServiceManagerConfig` wird zudem die entsprechende Konfiguration aus der zuvor geladenen `application.config.php` übergeben.

Der `ServiceManager` ist eine Art `Zend_Registry` mit erweiterten Funktionen. Während die `Zend_Registry` der Version 1 lediglich existierende Objekte unter einem bestimmten Key ablegen und später wieder laden konnte (Key-Value-Storage), geht der `ServiceManager` einige Schritte weiter.

Services und Service-Erzeugung

Neben der Verwaltung von Services in Form von Objekten, kann für einen Key auch ein “Erzeuger” registriert werden, der den jeweiligen Service bei Bedarf zunächst erstellt. Dazu lassen sich entweder Klassen (vollqualifiziert, also ggf. unter Angabe des Namespaces) angeben, die dann auf Wunsch instanziiert werden, etwa in der Form

```
1 <?php
2 $serviceManager->setInvokableClass(
3     'MyService',
4     'Helloworld\Service\MyService'
5 );
```

Listing 6.3

oder es lassen sich Factories hinterlegen:

```
1 <?php
2 $serviceManager->setFactory(
3     'MyServiceFactory',
4     'Helloworld\Service\MyServiceFactory'
5 );
```

Listing 6.4

Damit der ServiceManager mit der MyServiceFactory umgehen kann, muss diese das FactoryInterface implementieren, dass eine Methode createService erfordert. Diese Methode wird dann vom ServiceManager aufgerufen. Als leichtgewichtige Implementierung einer Factory kann auch direkt eine Callback-Funktion übergeben werden:

```
1 <?php
2 $serviceManager->setFactory(
3     'MyServiceFactory',
4     function($serviceManager) {
5         // [...]
6     }
7 );
```

Listing 6.5

Alternativ kann analog auch eine abstrakte Factory hinterlegt werden, wobei diese ohne Identifier hinzugefügt wird, hier exemplarisch in einer Callback-Variante:

```
1 <?php
2 $serviceManager->addAbstractFactory(
3     function($serviceManager) {
4         // [...]
5     }
6 );
```

Listing 6.6

Zusätzlich können sog. “Initializer” hinterlegt werden, die dafür sorgen, dass ein Service bei Abruf mit Werten oder Referenzen zu anderen Objekten ausgestattet wird. “Initializer” können so konzipiert werden, dass Sie Objekte in einen Service injizieren, wenn der jeweilige Service eine definierte Schnittstelle implementiert. Wir werden uns das später noch einmal in Aktion ansehen. Für den Moment merken wir uns nur, dass es sie gibt. Alle vom ServiceManager bereitgestellten Services sind sog. “shared services”, das heißt, dass die Instanz eines Services - bei Bedarf erzeugt oder bereits vorhanden - auch bei der zweiten Anforderung direkt zurückgegeben wird. Die Instanz des Services wird also wiederverwendet. Dies gilt für alle durch das Framework definierten Standard-Services; einzige Ausnahme ist dabei der EventManager. Wird ein neuer Service registriert, kann man ebenfalls explizit die Wiederverwendung von Instanzen unterbinden:

```
1 <?php
2 $serviceManager->setInvokableClass(
3     'myService',
4     'Helloworld\Service\MyService',
5     false
6 );
```

Listing 6.7

Standard-Services, Teil 1

Die ServiceManagerConfig sorgt ganz zu Anfang der Requestverarbeitung dafür, dass eine Reihe von Standard-Services bereitgestellt werden. Wichtig ist die Tatsache, dass sich im ServiceManager dann Services befinden, die teilweise nur vom Framework (oder besser dessen MVC-Implementierung) genutzt werden, während einige andere Services auch für den Anwendungsentwickler, etwa im Kontext eines Controllers, hilfreich sind. Das wird später noch klarer.

Invokables

Der folgende Service wird als “Invokable”, also durch Angabe einer Klasse, die bei Bedarf dann instanziiert wird, verfügbar gemacht:

- `SharedEventManager` (`Zend\EventManager\SharedEventManager`): Erlaubt die Registrierung von Listnern für bestimmte Events, auch, wenn der dazu notwendige Event-Manager noch nicht zur Verfügung steht. Der `SharedEventManager` wird automatisch einem neuen `EventManager` verfügbar gemacht, wenn dieser über den `ServiceManager` erzeugt wird. Weitere Erklärungen zum `SharedEventManager` finden sich später im Buch.

Factories

Factories dienen im Kontext des `ServiceManager` dazu, Services verfügbar zu machen, die bis zur eigentlichen Anforderung noch nicht existent sind, sondern eben durch eine Factory “on demand” gebaut werden. Die folgenden Services werden über den Umweg einer Factory standardmäßig bereitgestellt:

- `EventManager` (`Zend\Mvc\Service\EventManagerFactory`): Der `EventManager` kann Ereignisse erzeugen und registrierte Listener darüber informieren. Lässt sich auch über den Alias `Zend\EventManager\EventManagerInterface` anfordern.
- `ModuleManager` (`Zend\Mvc\Service\ModuleManagerFactory`): Verwaltet die Module einer ZF2-Anwendung.

Konfiguration

Der `ServiceManager` wird für die Nutzung im Rahmen der Requestverarbeitung also maßgeblich über zwei Konfigurationen gesteuert: Die `ServiceManagerConfig`, die eine Reihe von Standard-services für die Verarbeitung von Requests definiert, als auch durch die `application.config.php` bzw. die Modul-spezifischen Konfigurationen. Maßgeblich dafür ist jeweils der Key `service_manager`:

```
1  <?php
2  return array(
3      // [...]
4      'service_manager' => array(
5          // [...]
6      ),
7  ),
8  // [...]
9  );
```

Listing 6.8

Unterhalb des Keys `service_manager` sind dann die folgenden Schlüssel möglich:

- `services`: Definition von Services mithilfe bereits instanzierter Objekte.

- `invokables`: Definition von Services durch Angabe einer Klasse, die bei Bedarf instanziiert wird.
- `factories`: Definition von Fabriken, die Services instanziiieren.
- `abstract_factories`: Definition abstrakter Fabriken.
- `aliases`: Definition von Aliasen.
- `shared`: Ermöglicht die explizite Angabe, ob ein bestimmter Service mehrfach verwendet oder jeweils bei erneuten Bedarf frisch instanziiert werden soll.

Die `application.config.php` wird, sobald der `ServiceManager` zur Verfügung steht, dann in Gänze, also auch samt den anderen, nicht-`ServiceManager` relevanten Abschnitten, noch selbst als Service verfügbar gemacht:

```
1 <?php
2 // [...]
3 $serviceManager->setService('ApplicationConfig', $configuration);
```

Listing 6.9

Das ist wichtig, greifen doch andere Komponenten im Rahmen der Request-Verarbeitung, etwa der `ModuleManager` oder `ViewManager`, auf diesen Service zu.

Zum jetzigen Zeitpunkt steht also der `ServiceManager`, ausgestattet mit diversen Standard-Services Gewehr bei Fuß und wartet sozusagen nur darauf, dass es endlich los geht. Denn bisher ist eigentlich noch nicht viel passiert, außer einiger grundlegender Vorbereitungen. Tatsächlich existieren an dieser Stelle fast alle der oben genannten Services noch nicht, weil sie noch gar nicht angefordert wurden und erst bei Bedarf erzeugt werden.

6.2 Einen eigenen Service schreiben

Erstellen wir doch für unser Helloworld-Modul exemplarisch mal einen eigenen Service. Dazu legen wir zunächst ein weiteres Unterverzeichnis `Service` im `src/Helloworld`-Verzeichnis unseres Moduls an:

```
1  Module.php
2  config/
3      module.config.php
4  public/
5      images/
6      css/
7      js/
8  src/
9      Helloworld/
10          Controller/
11              IndexController.php
12          Service/
13              GreetingService.php
14  view/
15      Helloworld/
16          Index/
17              index.phtml
```

Dort erstellen wir eine Klasse `GreetingService`. Diese Klasse muss keine besonderen Schnittstellen implementieren oder von etwaigen Basisklassen ableiten, ist also ein sog. “POPO”, ein “Plain Old PHP Object”. Wichtig ist lediglich, dass wir daran denken, die Klasse im richtigen Namespace verfügbar zu machen:

```
1  <?php
2  namespace Helloworld\Service;
3
4  class GreetingService
5  {
6      public function getGreeting()
7      {
8          if(date("H") <= 11)
9              return "Good morning, world!";
10         else if (date("H") > 11 && date("H") < 17)
11             return "Hello, world!";
12         else
13             return "Good evening, world!";
14     }
15 }
```

Listing 6.10

Service als Invokable zur Verfügung stellen

Um diese Klasse als Service in unserem Controller zu verwenden und eine Uhrzeit-bezogene Begrüßung anzeigen zu können, müssen wir die Klasse dem `ServiceManager` als Service hinzufügen. Dies können wir im Rahmen unseres Moduls auf zwei Arten machen: Im Zuge der Module-Konfiguration (`module.config.php`) durch Hinzunahme des Abschnitts

```
1  <?php
2  // [...]
3  'service_manager' => array(
4      'invokables' => array(
5          'greetingService' => 'Helloworld\Service\GreetingService'
6      )
7  )
8  // [...]
```

Listing 6.11

oder programmatisch durch Hinzunahme der Funktion `getServiceConfig()` in der `Module.php`:

```
1  <?php
2  public function getServiceConfig()
3  {
4      return array(
5          'invokables' => array(
6              'greetingService'
7                  => 'Helloworld\Service\GreetingService'
8          )
9      );
10 }
```

Listing 6.12

Beide Wege führen zum Ziel. Unser Service ist nun in Form eines “invokables” bereitgestellt. Wir können im `IndexController` unseres `Helloworld`-Moduls den Service anfordern und einsetzen:

```
1  <?php
2  // [...]
3  public function indexAction()
4  {
5      $greetingSrv = $this->getServiceLocator()
6          ->get('greetingService');
7
8      return new ViewModel(
9          array('greeting' => $greetingSrv->getGreeting())
10     );
11 }
```

Listing 6.13

Controller über eine Factory-Klasse zur Verfügung stellen

Ein Problem haben wir jetzt allerdings: Der Controller ist abhängig von einem Service (und dem ServiceManager), den er sich aktiv holt. Er instanziert die Klasse zwar nicht selbst, dass ist schon einmal gut, bietet sich doch so die Möglichkeit für uns, ggf. eine alternative Implementierung im ServiceManager bereitzustellen, aber dennoch sorgt er sich selbst aktiv darum, dass alle Abhängigkeiten aufgelöst sind. Das macht uns spätestens dann Probleme, wenn wir den Controller “unittesten” wollen. Ein möglicher Ausweg wäre hier das zuvor bereits angesprochene “Dependency Injection” bzw. “Inversion Of Control”: Notwendige Kollaborateure werden dabei automatisch bereitgestellt und müssen nicht mehr aktiv angefordert werden. Im Rahmen des ServiceManager können wir dieses Vorgehen etwa über eine vorgeschaltete “Factory” realisieren.

Dazu erstellen wir im gleichen Verzeichnis, in dem auch der IndexController abgelegt ist, die “Factory” IndexControllerFactory:

```
1  <?php
2  namespace HelloWorld\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class IndexControllerFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $ctr = new IndexController();
12
13         $ctr->setGreetingService(
```



```
14         $serviceLocator->getServiceLocator()  
15         ->get('greetingService')  
16     );  
17  
18     return $ctr;  
19 }  
20 }
```

Listing 6.14

Zudem ändern wir die `module.config.php` im Abschnitt `controllers` wie folgt:

```
1  <?php  
2  // [...]  
3  'controllers' => array(  
4      'factories' => array(  
5          'Helloworld\Controller\Index'  
6              => 'Helloworld\Controller\IndexControllerFactory'  
7      )  
8  )  
9  // [...]
```

Listing 6.15

Von nun an wird also unser `IndexController` nicht mehr durch Instanzieren einer definierten Klasse erzeugt, sondern über die hinterlegte Factory, die zuvor die Kollaborateure des Controllers organisiert und dem Controller in diesem Fall via “Setter Injection” zur Verfügung stellt. Im `IndexController` müssen wir dazu noch den entsprechenden “Setter” hinzufügen und in der eigentlichen Action dann auf die entsprechende Membervariable, statt auf den `ServiceLocator` zugreifen:

```
1  <?php  
2  
3  namespace Helloworld\Controller;  
4  
5  use Zend\Mvc\Controller\AbstractActionController;  
6  use Zend\View\Model\ViewModel;  
7  
8  class IndexController extends AbstractActionController  
9  {  
10     private $greetingService;  
11  
12     public function indexAction()  
13     {  
14         // ...  
15     }  
16 }
```

```
13     {
14         return new ViewModel(
15             array(
16                 'greeting' => $this->greetingService->getGreeting()
17             )
18         );
19     }
20
21     public function setGreetingService($service)
22     {
23         $this->greetingService = $service;
24     }
25 }
```

Listing 6.16

Eine Factory-Klasse lässt sich also sowohl für die Erzeugung eines Services, als auch für die Erzeugung eines Controllers einsetzen, weil beide dazu eine Instanz eines “ServiceManager” verwenden. Es ist nämlich wie folgt: Der `Zend\ServiceManager` kommt im ZF2 gleich mehrfach zum Einsatz. Einmal in der Form, in der wir ihn bereits kennengelernt haben: Als zentrale Instanz, über die initial sogar die Application selbst, also der “Container” der gesamten Anwendung, wenn man denn so will, erzeugt wird. Und dann gibt es, wie wir auch gleich noch ausführlicher diskutieren werden, eine Reihe spezialisierter “ServiceManager”, etwa einen nur für die Controller der Anwendung. Interessant sind in diesem Zuge insbesondere die folgende Zeilen der `IndexControllerFactory`:

```
1  <?php
2  public function createService(ServiceLocatorInterface $serviceLocator)
3  {
4      // [...]
5      $ctr->setGreetingService(
6          $serviceLocator->getServiceLocator()
7              ->get('greetingService')
8      );
9      // [...]
10 }
11 // [...]
```

Listing 6.17

Es fällt auf, dass auf dem `$serviceLocator` zunächst `getServiceLocator()` aufgerufen wird. Der Grund dafür liegt in der Tatsache, dass der Methode `createService()` der Factory jeweils immer der “ServiceManager” übergeben wird, der mit der Erzeugung des Services beauftragt wurde, in diesem Fall also der für die Erzeugung von Controllern abgestellte `ControllerLoader` (wird automatisch

vom Framework angesprochen). Dieser wiederum hat aber keinen Zugriff auf den zuvor von uns erstellten GreetingService, den wir nur im “zentralen ServiceManager” verfügbar gemacht haben (er ist ja letztlich auch kein Controller). Damit an dieser Stelle aber nun trotzdem die Services des “zentralen ServiceManager” in Anspruch genommen werden können, greift der ControllerLoader über getServiceLocator() auf den “zentralen ServiceManager” zu, der ihm über ebenjene Methode zur Verfügung gestellt wird. Doch mehr zu den Details dieses Mechanismus etwas später in diesem Kapitel.

Controller über Factory-Callback zur Verfügung stellen

Mit der IndexControllerFactory haben wir nun allerdings eine zusätzliche Klasse in unserem Sourcecode. Das ist grundsätzlich erst einmal kein Problem, könnte in einem Fall wie diesem, in dem es kaum eine herausfordernde Aufgabe bei der Erzeugung für die Factory gibt, aber vielleicht etwas zu viel des Guten sein. Eine leichtgewichtige Alternative, die es uns ebenfalls ermöglicht, die direkte Abhängigkeit zu vermeiden, ist die Nutzung einer Callback-Funktion als Factory in der module.config.php:

```
1  <?php
2  // [...]
3  'controllers' => array(
4      'factories' => array(
5          'Helloworld\Controller\Index' => function($serviceLocator) {
6              $ctr = new Helloworld\Controller\IndexController();
7
8              $ctr->setGreetingService(
9                  $serviceLocator->getServiceLocator()
10                     ->get('greetingService')
11             );
12
13             return $ctr;
14         }
15     )
16 )
17 // [...]
```

Listing 6.18

Der Code zur Erzeugung des IndexController, der sich bislang in der IndexControllerFactory befunden hat, ist nun direkt in die module.config.php gewandert.

Service über Zend\Di zur Verfügung stellen

Egal welche Form von Factory verwendet wird, alle haben gemeinsam, dass die Erzeugung des jeweiligen Objekts programmatisch erfolgt, also entsprechender PHP-Code geschrieben werden muss. Eine alternative Möglichkeit stellt Zend\Di dar, mit dessen Hilfe wir ganze Objektgraphen über Konfigurationsdateien erzeugen können. Dazu später mehr.

6.3 ModuleManager

Doch nun zurück zur Request-Verarbeitung: Nachdem der ServiceManager soweit vorbereitet ist, wird er auch schon bereits das erste Mal eingesetzt, um den ModuleManager über die registrierte Factory zu erzeugen, bevor dann das Laden der Module angestoßen wird:

```
1 <?php
2 $serviceManager->get('ModuleManager')->loadModules();
```

Listing 6.19

Erzeugen des ModuleManagers

Die ModuleManagerFactory hat die Aufgabe, den Service ModuleManager zu erzeugen. Wir erinnern uns: Eine Factory kommt immer dann zum Einsatz, sobald die Erzeugung eines Objektes komplexer wird. Im Rahmen der Erzeugung wird zunächst einmal ein neuer EventManager über den ServiceManager angefordert und dem ModuleManager zur Verfügung gestellt. Der ModuleManager ist so also in der Lage, Ereignisse zu erzeugen und zuvor registrierte “Listener” zu informieren. Die folgenden Ereignisse werden vom ModuleManager (zu einem späteren Zeitpunkt!) ausgelöst:

- `loadModules`: Tritt ein, wenn die Module geladen werden.
- `loadModule.resolve`: Tritt für jedes zu ladende Modul ein, wenn die notwendigen Dateien eingelesen werden.
- `loadModule`: Tritt beim Laden eines Moduls für jedes Modul ein, wenn die eingelesenen Dateien ausgeführt werden.
- `loadModules.post`: Tritt ein, nachdem alle Module geladen wurden.

Modulbezogene Listener

Die ModuleManagerFactory macht allerdings noch eine ganze Menge mehr. Zunächst einmal erzeugt sie eine ganze Reihe von Listenern, die für die oben genannten Events registriert werden:

- `ModuleAutoloader`: Stellt sicher, dass die Klasse `Module` der einzelnen Module automatisch geladen werden kann.
- `ModuleResolverListener`: Instanziert die `Module.php` des jeweiligen Moduls.
- `AutoloaderListener`: Ruft in `Module` die Methode `getAutoloaderConfig()` auf, um Informationen darüber zu gewinnen, wie die Klassen des Moduls automatisch geladen werden können.
- `OnBootstrapListener`: Schaut nach, ob `Module` über eine Methode `onBootstrap()` verfügt und registriert den Aufruf dieser Methode für das `bootstrap` - Ereignis, dass zu einem späteren Zeitpunkt von der `Application` ausgelöst wird.
- `InitTrigger`: Schaut nach, ob `Module` über die Methode `init()` verfügt. Falls ja, wird sie aufgerufen.
- `ConfigListener`: Schaut nach, ob `Module` über eine Methode `getConfig()` verfügt, die, falls vorhanden, aufgerufen und das zurückgelieferte Modul-Konfigurations-Array mit den anderen Konfigurationen vereint.
- `LocatorRegistrationListener`: Stellt sicher, dass Instanzen aller `Module`-Klassen, die das `ServiceLocatorRegisteredInterface` implementieren, der `ServiceManager` injiziert wird.
- `ServiceListener`: Ruft falls vorhanden die Methoden `getServiceConfig()`, `getControllerConfig()`, `getControllerPluginConfig()`, `getViewHelperConfig()` in der `Module`-Klasse auf (bzw. liest die entsprechenden Konfigurationen aus; weitere Details dazu im Folgenden), verarbeitet die zusammengeführten Konfigurationen aller Module, wendet sie auf den `ServiceManager` an und ergänzt diesen zudem um weitere Standard-Services.

Standard-Services, Teil 2

Nachdem bereits im Zuge der `ServiceManager`- Erzeugung einige Standard-Services bereitgestellt wurden, darunter neben dem `EventManager` ja auch der `ModuleManager` selbst, registriert der `ServiceListener` (auf Wunsch von dessen `Factory`) zudem einen bunten Strauß zusätzlicher Services, die im Zuge der Request-Verarbeitung erforderlich sind. An dieser Stelle ein kurzer Hinweis: Zu diesem Zeitpunkt kann und soll die Funktionsweise eines jeden Services noch nicht voll verstanden sein. Viele der Services, die hier vom `ModuleManager` registriert werden, laufen uns im weiteren Verlauf dieses Kapitels oder Buches noch einmal über den Weg. Die folgende Aufstellung dient also viel mehr als Referenz und Ausblick, auf das, was da noch kommt. Die folgenden Services werden also - hier nach Art und Weise der Registrierung aufgeschlüsselt - registriert:

Invokables

- `RouteListener` (`Zend\Mvc\RouteListener`): Lauscht später auf das Mvc-Ereignis `onRoute` und sorgt dann dafür, dass der Router mit der Auflösung auf den passenden Controller beauftragt wird.
- `DispatchListener` (`Zend\Mvc\DispatchListener`): Lauscht später auf das Mvc-Ereignis `onDispatch` und sorgt dann dafür, dass der `ControllerLoader` den zuvor auserwählten Controller lädt und ausführt.

Factories

- **Application** (Zend\Mvc\Service\ApplicationFactory): Die Application (durch die hinterlegte Factory erzeugt) repräsentiert, wenn man so will, die gesamte Verarbeitungskette und überhaupt die gesamte Anwendung.
- **Configuration** (Zend\Mvc\Service\ConfigFactory): Der erzeugte Service Config gibt die zusammengeführte Konfiguration für die Anwendung zurück. Steht auch über den Alias Config zur Verfügung.
- **ConsoleAdapter** (Zend\Mvc\Service\ConsoleAdapterFactory): Service zum Zugriff auf die Kommandozeile.
- **DependencyInjector** (Zend\Mvc\Service\DiFactory): Das Zend Framework kommt mit einer eigenen Implementierung des sog. “Dependency Injection”, mit dessen Hilfe komplexe Objektgraphen auf Basis einer umfassenden Konfiguration automatisch “zusammengesteckt” werden. Anstelle des Keys DependencyInjector kann man auch seine Aliase Di oder Zend\Di\LocatorInterface verwenden. Zend\Di sehen wir uns zu einem späteren Zeitpunkt noch einmal im Detail an.
- **Router, HttpRouter, ConsoleRouter** (Zend\Mvc\Service\RouterFactory): Der von der Factory erzeugte Router - Service ermittelt auf Basis der Request-URL den aufzurufenden Controller, ggf. auch im “Kommandozeilenmodus”.
- **Request** (Zend\Mvc\PhpEnvironment\Request): Bietet Zugriff auf sämtliche Requestinformationen, u.a. die Requestparameter.
- **Response** (Zend\Http\PhpEnvironment\Response): Repräsentiert die im Laufe der Verarbeitung erzeugte Antwort an den Client.
- **ViewManager**: Das, was der ModuleManager für die Module und der ServiceManager für die Services ist, ist der ViewManager für die Verwaltung der Views und dessen Verarbeitung. Er kümmert sich darum, dass aus den Daten irgendwann etwa Webseiten mit HTML-Markup werden.
- **ViewJsonRenderer** (Zend\Mvc\Service\ViewJsonRendererFactory): Ermöglicht die Realisierung von [RESTful¹](http://de.wikipedia.org/wiki/Representational_State_Transfer)-Controllern und damit von Webservices, die dem REST-Architekturstil folgen. Diesem Thema ist in dem Buch ein eigenes Kapitel gewidmet.
- **ViewJsonStrategy** (Zend\Mvc\Service\ViewJsonStrategyFactory): Sorgt dafür, dass der ViewJsonRenderer aufgerufen wird, wenn erforderlich. Im Rahmen dieser Strategy wird etwa überprüft, ob das vom Controller zurückgegebene ViewModel vom Typ JsonModel ist.
- **ViewFeedRenderer** (Zend\Mvc\Service\ViewFeedRendererFactory): Ermöglicht die Realisierung von RSS- oder Atom-Feeds auf Basis der von einem Controller zurückgegebenen View-Daten.
- **ViewFeedStrategy** (Zend\Mvc\Service\ViewFeedStrategyFactory): Sorgt dafür, dass der ViewFeedRenderer aufgerufen wird, wenn erforderlich. Teil dieser Strategy ist die Feststellung, ob das vom Controller zurückgegebene ViewModel vom Typ FeedModel ist.
- **ViewResolver** (Zend\Mvc\Service\ViewResolverFactory): Macht View-Templates auffindbar.

¹http://de.wikipedia.org/wiki/Representational_State_Transfer

- `ViewTemplateMapResolver` (`Zend\Mvc\Service\ViewResolverFactory`): Macht View-Templates für den `ViewResolver` auf Basis einer Map auffindbar.
- `ViewTemplatePathStack` (`Zend\Mvc\Service\ViewTemplatePathStackFactory`): Macht View-Templates für den `ViewResolver` auf Basis einer Liste von Pfaden auffindbar.

Als auch zusätzlich:

- `ControllerLoader` (`Zend\Mvc\Service\ControllerLoaderFactory`): Der `ControllerLoader` ist in der Lage dazu einen Controller zu laden, der zuvor im Rahmen des Routings ermittelt wurde.
- `ControllerPluginManager` (`Zend\Mvc\Service\ControllerPluginManagerFactory`): Stellt den `ControllerPluginManager` und damit eine Reihe von Plugins bereit, die in Controllern verwendet werden können. Darunter etwa das Plugin `redirect`, über das sich eine Weiterleitung realisieren lässt. Der Service kann auch über die Keys `ControllerPluginBroker`, `Zend\Mvc\Controller\PluginBroker` oder `Zend\Mvc\Controller\PluginManager` angefordert werden.
- `ViewHelperManager` (`Zend\Mvc\Service\ViewHelperManagerFactory`): Erzeugt den `ViewHelperManager`, der die Verwaltung sog. “View Helper” verantwortet.

Die letzten drei Services sind besonders interessant, stellen sie ihrerseits doch wieder selbst neue “ServiceManager”, im ZF-Jargon “Scoped ServiceManager” genannt, dar. Puh, jetzt wird es langsam kompliziert! Also noch einmal langsam der Reihe nach. Halten wir zunächst einmal fest: Im System gib es den einen “zentralen ServiceManager”. Alle wesentlichen “Anwendungsservices”, werden über ihn erzeugt. Er ist sowohl technisch ein `ServiceManager`, als auch für uns konzeptionell “der zentrale ServiceManager”. Nun gibt es aber bestimmte Services, die der `ServiceManager` nicht selbst zur Verfügung stellt, sondern spezialisierte “Sub-ServiceManager” bzw. “Scoped ServiceManager”, die es ebenfalls ermöglichen, Services über die bekannten Mechanismen, also “Invokables”, “Factories”, usw., bereitzustellen. Auch sie sind technisch alle `ServiceManager`.

Schauen wir dazu noch einmal auf das letzte Kapitel zurück, in dem wir einen eigenen Controller geschrieben haben. Dort finden wir in der `module.config.php` folgende Passage:

```
1  <?php
2  // [...]
3  'controllers' => array(
4      'invokables' => array(
5          'Helloworld\Controller\Index'
6              => 'Helloworld\Controller\IndexController'
7      )
8  )
9  // [...]
```

Listing 6.20

Wenn dieses Stückchen Konfiguration interpretiert wird, führt dies dazu, dass unter dem Key `HelloWorld\Controller\Index` auf die passende Controller-Klasse verwiesen wird, die als “Invokable” im `ControllerLoader`, einem der standardmäßigen “Scoped ServiceManager”, registriert wird. Wird dieser Controller also später im Rahmen des Routings als passend identifiziert und muss dann instanziiert werden, so bedient sich das System dazu des `ControllerLoader`. In diesem Kontext kann man einen Controller dann ebenfalls als einen “Service” bezeichnen.

Dieses Vorgehen der spezialisierten “Sub-ServiceManager” für bestimmte Arten von Services hat einige Vorteile. Etwa wird auf diese Art und Weise der “zentrale ServiceManager” für die “Application Services” selbst nicht mit unzähligen Services vollgestopft und es lassen sich leicht alle im System vorhandenen Controller ermitteln, was andernfalls gar nicht so einfach wäre. Hier noch einmal die verschiedenen “ServiceManager” auf einen Blick:

- Application Services (`Zend\ServiceManager\ServiceManager`): Konfiguration über Key `service_manager` oder Methode `getServiceConfig()` (definiert in `ServiceProviderInterface`).
- Controllers (`Zend\Mvc\Controller\ControllerManager`): Konfiguration über Key `controllers` oder Methode `getControllerConfig()` (definiert in `ControllerProviderInterface`). Ist im “zentralen ServiceManager” über die Service-Bezeichnung `ControllerLoader` zu beziehen.
- “Controller Plugins” (`Zend\Mvc\Controller\PluginManager`): Konfiguration über Key `controller_plugins` oder Methode `getControllerPluginConfig()` (definiert in `ControllerPluginProviderInterface`). Ist im “zentralen ServiceManager” über die Service-Bezeichnung `ControllerPluginManager` zu beziehen.
- “View Helpers” (`Zend\View\HelperPluginManager`): Konfiguration über Key `view_helpers` oder Methode `getViewHelperConfig()` (definiert in `ViewHelperProviderInterface`). Ist im “zentralen ServiceManager” über die Service-Bezeichnung `ViewHelperManager` zu beziehen.

Laden der Module

Nachdem der `ModuleManager` vorbereitet ist und die notwendigen Listener registriert sind, kommt das eigentliche Laden der Module durch Aufruf der Methode `loadModules()` des `ModuleManager`. Hier passiert jetzt eigentlich ziemlich wenig, findet die tatsächliche Verarbeitung, etwa das Aufrufen der oben genannten Methoden der `Module.php`, doch in den vielen registrierten Listnern statt. Zunächst wird der Event `loadmodules.pre` getriggert, dann für jedes aktivierte Modul das Event `loadModule.resolve` und `loadModule`. Zum Abschluss wird noch einmal der Event `loadModules.post` ausgelöst. Und das war es auch schon.

Das Module-Event-Objekt

Ein Konzept des EventManager ist, dass es neben dem Auslöser eines Events und den dafür registrierten Listnern (“Empfängern”) noch das Event selbst - als eigenständiges Objekt repräsentiert

- gibt. Es wird allen Listenern zur Verfügung gestellt. Dieses Objekt dient dazu, Event-relevante Zusatzinformationen zu übergeben, etwa einem Hinweis darauf, an welcher Stelle im Code der Event ausgelöst wurde. Zudem können weitere Daten übermittelt werden, die für die Event-Verarbeitung in den Listenern hilfreich sind. So ist für einen Listener beim `loadModule` in aller Regel von Interesse, um welches Modul es sich denn konkret handelt, dass gerade geladen wird.

Um der Tatsache gerecht zu werden, dass je nach Kontext des Events andere Daten von Interesse sind, erlaubt es der `EventManager` des Zend Framework 2, situationsabhängig spezielle Event-Klassen zu hinterlegen. Für das Event-Prinzip im Rahmen des `ModuleManager` gibt es dazu die spezielle Klasse `ModuleEvent`, die etwa sowohl das betroffene Modul als auch zusätzlich den Namen des Moduls mit sich führt.

Aktivierung eines Moduls

Damit ein Modul vom `ModuleManager` überhaupt berücksichtigt wird, ist eine explizite Aktivierung in der `application.config.php` im Verzeichnis `config` erforderlich:

```
1 <?php
2 return array(
3     'modules' => array(
4         'Application',
5         'Helloworld'
6     )
7 );
```

Listing 6.21

Methoden der Module-Klasse

Wie wir gesehen haben, werden eine ganze Reihe von Methoden der `Module`-Klasse eines Moduls aufgerufen, wenn wir sie denn implementiert haben. Für das Framework gibt es zwei maßgebliche Möglichkeiten um herauszufinden, ob dies der Fall ist: Entweder, die `Module`-Klasse implementiert ein bestimmtes Interface (das kann ja via `instanceof` geprüft werden), oder aber es wird einfach so die jeweilige Methode implementiert (kann via `method_exists()` - Aufruf überprüft werden).

Werfen wir noch einmal einen detaillierten Blick auf die Methoden der Klasse `Module`, die automatisch vom Framework aufgerufen und vom Anwendungsentwickler genutzt werden können:

- `getAutoloadingConfig()` (definiert in `AutoloaderProviderInterface`): Diese Methode hatten wir bereits in unserem “Helloworld-Modul” erstellt. Sie liefert Informationen darüber, wie die Klassen des Moduls automatisch geladen werden können. Lassen wir diese Methode weg, können in aller Regel die Klassen des Moduls, unter anderem dessen Controller, nicht

geladen werden und bei Aufruf der entsprechende URL gibt es ernsthafte Probleme. Es sollte also immer sichergestellt werden, dass dem Framework Angaben dazu gemacht werden, wie die Klassen automatisch geladen werden können. Rein technisch nimmt das Framework übrigens die Informationen dazu, eine passende Loader-Implementierung für dieses Modul via `spl_autoload_register()` einzubinden.

- `init()` (definiert in `InitProviderInterface`): Diese Methode erlaubt es dem Anwendungsentwickler, das eigene Modul zu initialisieren, also etwa eigene Listener für bestimmte Events zu registrieren. Die Methode bekommt bei Bedarf den `ModuleManager` übergeben und kann so auf die entsprechenden Events (des `ModuleManager`) oder auf die Module zugreifen. Wichtig ist die Tatsache, dass diese Methode immer und bei jedem Request aufgerufen wird - und zwar für jedes Modul. Man sollte sich also bewusst machen, dass hier ein guter Ort ist, um die Ladezeiten einer Anwendung zu ruinieren. Im Rahmen der Methode `init()` sollten also nur sehr wenige und im besten Fall leichtgewichtige Operationen vorgenommen werden. Versucht man, die Geschwindigkeit einer ZF2-Anwendung zu verbessern, sollte man sich also immer zuerst die `init()` - Methoden der aktivierten Module ansehen.

Hier exemplarisch ein Beispiel für die Nutzung der Methode `init()`:

```
1  <?php
2  namespace HelloWorld;
3
4  use Zend\ModuleManager\ModuleManager;
5  use Zend\ModuleManager\ModuleEvent;
6
7  class Module
8  {
9      public function init(ModuleManager $moduleManager)
10     {
11         $moduleManager->getEventManager()
12             ->attach(
13                 ModuleEvent::EVENT_LOAD_MODULES_POST,
14                 array($this, 'onModulesPost')
15             );
16     }
17
18     public function onModulesPost()
19     {
20         die("Alle Module sind geladen!");
21     }
22
23     // [...]
24 }
```

Listing 6.21

- `onBootstrap()` (definiert in `BootstrapListenerInterface`): Eine weitere Möglichkeit für den Anwendungsentwickler, ein modulespezifisches Bootstrapping vorzunehmen. Im Grunde hat diese Methode den gleichen Sinn und Nutzen wie `init()`, allerdings wird `onBootstrap` zu einem späteren Zeitpunkt der Verarbeitung aufgerufen, nämlich dann, wenn der `ModuleManager` bereits seine Arbeit erledigt und das Ruder an die `Application` übergeben hat. Innerhalb der `onBootstrap()` - Methode sind also ggf. Services und Daten verfügbar, die in der `init()` noch nicht zugänglich waren.
- `getConfig()` (definiert in `ConfigProviderInterface`): Auch diese Methode ist uns bereits bekannt. Sie bietet die Möglichkeit, auf die modulspezifische Konfigurationsdatei zu verweisen, die per Konvention `module.config.php` benannt wird und im Modul im Unterzeichnis `config` abgelegt wird. Das ist allerdings keine Pflicht. Streng genommen muss es diese Methode nicht geben, damit ein Modul lauffähig ist, in der Praxis kommt man aber nicht um eine modulspezifische Konfigurationsdatei herum, die man über diese Methode dem Framework zugänglich macht. Das Framework lässt einem in Bezug auf die Konfiguration eine gewisse Flexibilität. So können entweder via `getConfig()` alle Konfigurationen über ein oder mehrere externe Files zur Verfügung gestellt werden, oder es können spezielle “Config-Methoden” in der `Module`-Klasse implementiert werden, die sich auf die im System vorhandenen “ServiceManager” beziehen, also auf den `ServiceManager`, `ControllerLoader`, `ViewHelperManager` und `ControllerPluginManager`.
- `getServiceConfig()`: Ermöglicht die Konfiguration des `ServiceManager` und ist äquivalent zum “Config-Array-Key” `service_manager` in der `module.config.php`.
- `getControllerConfig()`: Ermöglicht die Konfiguration des `ControllerLoader` und ist äquivalent zum “Config-Array-Key” `controllers` in der `module.config.php`.
- `getControllerPluginConfig()`: Ermöglicht die Konfiguration des `ControllerPluginManager` und ist äquivalent zum “Config-Array-Key” `controller_plugins` in der `module.config.php`.
- `getViewHelperConfig()`: Ermöglicht die Konfiguration des `ViewHelperManager` und ist äquivalent zum “Config-Array-Key” `view_helpers` in der `module.config.php`.

Hierzu exemplarisch noch ein Beispiel: Ein eigener `ViewHelper` (mehr zum Konzept der “View Helper” auf den nachfolgenden Seiten) lässt sich entweder im Rahmen der `module.config.php` auf die folgende Art bekannt machen

```
1 <?php
2 'view_helpers' => array(
3     'invokables' => array(
4         'displayCurrentDate' => 'Helloworld\View\Helper\DisplayCurrentDate'
5     )
6 )
```

Listing 6.22

oder aber in der `Module.php` mit Hilfe der entsprechenden Methode:

```
1  <?php
2  public function getViewHelperConfig()
3  {
4      return array(
5          'invokables' => array(
6              'displayCurrentDate'
7                  => 'Helloworld\View\Helper\DisplayCurrentDate'
8          )
9      );
10 }
```

Listing 6.23

6.4 Application

Jetzt, wo der `ServiceManager` mit den notwendigen Services ausgestattet ist und der `ModuleManager` die Module der Anwendung geladen hat, kann die eigentliche Application gestartet und die Request-Verarbeitung angestoßen werden. Dies passiert in 3 Schritten, teils in der Methode `init()` der Application selbst, teils in der `index.php`: Das “Hochfahren” der Anwendung (`bootstrap()`), gefolgt von der Ausführung (`run()`) und last but not least die Rückgabe des erzeugten Ergebnisses (`send()`):

```
1  <?php
2  $application = $serviceManager->get('Application');
3  $application->bootstrap();
4  $application->run();
5  $application->send();
```

Listing 6.24

Erzeugen der Application & Bootstrapping

Das Anwendungs-Objekt wird über die im `ServiceManager` registrierte Factory erzeugt. Im Rahmen der Erzeugung holt sich die Application eine Reihe von Standard-Services, darunter sowohl den Request, als Grundlage für die weitere Verarbeitung, sowie die Response, die es im Rahmen der Verarbeitung mit Leben zu füllen gilt. Zudem bekommt die Application eine Referenz zum `ModuleManager` und eine eigene Instanz des `EventManager` (der ja im `ServiceManager` so hinterlegt ist, dass er nicht geshared wird; es wird also eine neue Instanz dieses Services zurückgegeben). Letzterer sorgt - analog zum `ModuleManager` - dafür, dass auch die Application Events auslösen und Listener informieren kann. Die Application löst eine Reihe von Ereignissen im Rahmen der Verarbeitung aus:

- `bootstrap`: Tritt ein, wenn die Anwendung “hochgefahren” wird.
- `route`: Tritt ein, wenn für die URL ein Controller und eine Action ermittelt wird.
- `dispatch`: Tritt ein, wenn der ermittelte Controller identifiziert und aufgerufen wird.
- `render`: Tritt ein, wenn das Ergebnis für die Rückgabe auf Basis von Templates aufbereitet wird.
- `finish`: Tritt ein, wenn die Verarbeitung abgeschlossen wird.

Im Zuge des Bootstrapping registriert die `Application` (in diesem Fall übrigens tatsächlich die `Application` selbst, nicht die `ApplicationFactory`) auch gleich einige Listener, die sie ebenfalls über den `ServiceManager` bezieht: `RouteListener` für den Event `route`, `DispatchListener` für den Event `dispatch` und den `ViewManager` für einen bunten Strauß weiterer Listener, die sich um die Verarbeitung von Templates und Layouts kümmern. Doch dazu mehr im nächsten Abschnitt.

Weiterhin erstellt die `Application` dann noch das MVC-spezifische Event-Objekt (`MvcEvent`), das beim `EventManager` als Event-Objekt registriert wird. `MvcEvent` ermöglicht es dann den Listnern, auf `Request`, `Response`, `Application` und den Router zuzugreifen.

Dann wird das Ereignis `bootstrap` ausgelöst und die registrierten Listener werden ausgeführt. Dies können insbesondere auch die einzelnen Module der Anwendung sein, die sich für ebenjenes Event registriert haben.

Ausführung

Die Methode `run()`, die nach dem Bootstrapping ausgeführt wird, setzt dann eine Reihe der zuvor an Ort und Stelle platzierten Hebel in Bewegung. Zunächst einmal triggert die `Application` das Event `route`. Der zuvor für dieses Event registrierte `RouteListener` wird ausgeführt und der Router aus dem `MvcEvent` um seine Dienstleistung gebeten: Das Matching der URL auf eine definierte Route. Ein Route ist in diesem Fall die Beschreibung einer URL auf Basis eines definierten Musters. Wir gehen später noch im Detail auf die Mechanik des Routings ein. Für den Moment halten wir fest, dass der Router nun entweder eine Route findet, die auf die aufgerufene URL passt und damit den passenden Controller sowie die passende Action ermittelt hat, oder aber der Router kehrt mit schlechten Nachrichten zurück und hat keine Treffer erzielt. Bleiben wir hier zunächst auf dem Erfolgspfad: Die passende Route wird in Form eines `RouteMatch` - Objekts vom `RouteListener` im `MvcEvent` abgelegt. Der `MvcEvent` erweist sich also immer mehr als zentrales Objekt, in dem eine ganze Reihe wichtiger anderer Objekte und Daten verfügbar sind. Dann wird das `dispatch` - Ereignis ausgelöst, der `ControllerLoader` wird aufgerufen und dieser bedient sich des `MvcEvent` um den identifizierten und zu instanzierenden Controller zu erfahren. Dazu fordert der `DispatchListener` beim `ServiceManager` den `ControllerLoader` an (der ja technisch betrachtet für sich selbst genommen auch wieder ein “`ServiceManager`” ist) und bei letzterem dann den eigentlichen Controller. In diesem Zuge wird auch der Controller mit einem eigenen `EventManager` ausgestattet. Er kann nun also Ereignisse auslösen und Listener managen. Dann wird die Methode `dispatch()` des Controllers aufgerufen, die die weitere Verarbeitung übernimmt. Falls es zwischenzeitlich bei diesem Vorhaben irgendwelche Probleme gibt, wird das Ereignis `dispatch.error` ausgelöst, andernfalls wird das

Ergebnis des `dispatch()` - Aufrufs im `MvcEvent` abgelegt und zudem zurückgegeben und damit der Dispatch-Prozess innerhalb des Controllers abgeschlossen.

Controller sind im Zend Framework so konzipiert, dass sie nur über eine Methode `dispatch()` verfügen müssen. Diese wird von Außen aufgerufen, dabei das Request-Objekt übergeben und vom Controller erwartet, dass er ein Objekt vom Typ `Zend\Stdlib\ResponseInterface` zurückliefert, wenn die Arbeit getan ist. Damit das Prinzip von Controllers und Actions funktioniert, so wie man es gewohnt ist und es erwartet, muss diese Logik im Controller selbst implementiert werden, wird doch ansonsten immer nur die `dispatch`-Methode aufgerufen, nicht aber die entsprechende "Action"-Methode. Damit man dies nicht selbst erledigen muss, erbt der eigene Controller vom `Zend\Mvc\Controller\AbstractActionController`. Im Controller können dann nach beliebigen Actions angelegt werden, wenn man sich an die Konvention hält, dass der Methodenname auf "Action" enden muss:

```
1  <?php
2
3  namespace HelloWorld\Controller;
4
5  use Zend\Mvc\Controller\AbstractActionController;
6  use Zend\View\Model\ViewModel;
7
8  class IndexController extends AbstractActionController
9  {
10     private $greetingService;
11
12     public function indexAction()
13     {
14         return new ViewModel(
15             array(
16                 'greeting' => $this->greetingService->getGreeting(),
17                 'date' => $this->currentDate()
18             )
19         );
20     }
21 }
```

Listing 6.25

Zurück in der Application werden nun noch die beiden Ereignisse `render` und `finish` ausgelöst und die Methode `run()` beendet. Sehr interessant und hilfreich ist übrigens noch die folgende Tatsache: Normalerweise gibt eine "Action" am Ende der Verarbeitung ein Objekt vom Typ `ViewModel` zurück. Damit signalisiert es den nachfolgenden Verarbeitungsschritten implizit, dass das Ergebnis noch aufbereitet werden muss, bevor es zurückgegeben werden kann:

```
1  <?php
2  public function indexAction()
3  {
4      return new ViewModel(
5          array(
6              'greeting' => $this->greetingService->getGreeting(),
7              'date' => $this->currentDate()
8          )
9      );
10 }
```

Listing 6.26

Ein sehr praktisches Implementierungsdetail ist allerdings die Tatsache, dass, wenn anstelle eines ViewModel ein Response-Objekt zurückgegeben wird, die nachgelagerten render-Aktivitäten ausgelassen werden:

```
1  <?php
2  public function indexAction()
3  {
4      $resp = new \Zend\Http\PhpEnvironment\Response;
5      $resp->setStatusCode(503);
6      return $resp;
7  }
```

Listing 6.27

Dieser Mechanismus ist hilfreich, will man etwa kurzzeitig einen 503-Code zurückgeben, weil sich die Anwendung gerade in einer geplanten Wartung befindet, oder aber wenn man Daten eines bestimmten Mime-Types, etwa die Inhalte eines Bildes, PDF-Dokuments oder ähnlichem, zurückgeben will.

6.5 ViewManager

Bevor die Verarbeitung beendet ist, kommt der ViewManager noch ins Spiel. Im vorherigen Abschnitt hatten wir bereits gesehen, dass die Methode `onBootstrap()` im Rahmen des Bootstrappings der Application ausgeführt wird (weil sie für das entsprechende Ereignis registriert ist) und dort eine ganze Reihe weiterer Vorbereitungen getroffen werden. Bislang hatten wir das der Einfachheit halber ausgeblendet, nun blicken wir aber auch hier in die Details. Nach dem der ViewManager mit seinen vielen Verbündeten seine Arbeit erledigt hat sind wir auch tatsächlich einmal durch die ganze Verarbeitungskette durch.

Zunächst einmal holt sich der `ViewManager` vom `ServiceManager` die `Config`, die zu diesem Zeitpunkt bereits die zusammengefügte “Gesamtkonfiguration” der Anwendung und der Module darstellt. Der `ViewManager` schaut nach dem Key `view_manager` und verwendet die dort hinterlegte Konfigurationen. Dann beginnt das alte Spiel der Registrierung diverser Listener und der Bereitstellung zusätzlicher Services.

View-bezogene Listener

Die folgenden Listener werden erzeugt und allesamt an das Ereignis `dispatch` der `ActionController`-Klasse (oder genauer: des `EventManager` des `ActionController`) gehängt:

- `CreateViewModelListener`: Stellt sicher, dass nach Ausführung des Controllers immer ein Objekt vom Typ `ViewModel` für das Rendering bereitsteht, auch dann, wenn vom Controller nur `NULL` oder ein Array zur Verfügung gestellt wurde.
- `RouteNotFoundStrategy`: Erzeugt ein `ViewModel` für den Fall, dass kein Controller ermittelt werden und kein “View Model” erzeugt werden konnte (404-Fehler).
- `InjectTemplateListener`: Fügt das passende Template zum `ViewModel` für das nachfolgende Rendering hinzu.
- `InjectViewModelListener`: Fügt das `ViewModel` zum `MvcEvent` hinzu. Dieser Listener wird ebenfalls für das Ereignis `dispatch.error` der `Application` registriert, um auch im Fehlerfall ein `ViewModel` bereitstellen zu können.

Das Ereignis `dispatch` ist übrigens etwas gemein, gibt es jenes doch zwei Mal im System: Einmal wird es durch die `Application` ausgelöst, einmal durch den `ActionController`. Auch wenn die Bezeichnung des Ereignisses identisch ist (sie kann ja frei gewählt werden), so handelt es sich auf Grund der Tatsache, dass es von unterschiedlichen `EventManager`n ausgelöst wird, um zwei vollkommen unterschiedliche Ereignisse.

Übrigens existiert zu diesem Zeitpunkt der `EventManager` der jeweiligen konkreten Ausprägung des `ActionController` noch gar nicht, weil letzterer noch gar nicht erzeugt wurde. Diese Problem wird hier durch Nutzung des `SharedEventManager` umgangen. Mit Hilfe des `SharedEventManager` lassen sich Listener für Ereignisse eines `EventManager`s registrieren, den es zum Zeitpunkt der Registrierung noch gar nicht gibt. Für den Moment lassen wir es dabei. Wie genau dieser Mechanismus realisiert ist sehen wir uns im nächsten Kapitel genauer an.

View-bezogene Services

Zusätzlich werden eine Reihe View-bezogener Services im `ServiceManager` verfügbar gemacht:

- `View` und `View Model`: Da ist zunächst einmal die `View` mit seinem `View Model` selbst, die Repräsentation des aus einem Request für die Response erzeugten “Payloads”.

- `DefaultRenderingStrategy`: Hat Zugriff auf die View und wird für das Ereignis `render` der Application registriert. Tritt dieses Ereignis ein, wird der View das `ViewModel` übergeben, dass aus dem `MvcEvent` bezogen wird und dann die View aufgefordert, ebenjenes zu rendern.
- `ViewPhpRendererStrategy`: Das eigentliche Rendering übernimmt die View allerdings nicht selbst, sondern delegiert diese Aufgabe an die `ViewPhpRendererStrategy`, die zunächst den passenden Renderer festlegt und nach der Verarbeitung das fertige Ergebnis in die Response überführt.
- `RouteNotFoundStrategy`: Definiert das Verhalten im Falle einer 404-Situation.
- `ExceptionStrategy`: Definiert das Verhalten im Falle eines “Dispatch Errors”.
- `ViewRenderer`: Übernimmt die eigentliche Arbeit des Renderings, also der Verschmelzung der Daten des `ViewModel` und der passenden Templates.
- `ViewResolver`: Um das jeweilige Template zu lokalisieren greift der `ViewRenderer` auf den `ViewResolver` zurück.
- `ViewTemplatePathStack`: Ermöglicht es dem Resolver, ein Template auf Basis hinterlegter Pfade zu lokalisieren.
- `ViewTemplateMapResolver`: Ermöglicht es dem Resolver, ein Template auf Basis einer “Key-Value-Zuordnung” zu lokalisieren.
- `ViewHelperManager`: Ermöglicht es in Templates auf “ViewHelper” zurückzugreifen, die die Erzeugung von dynamischem Markup vereinfachen.

Im Falle eines Fehlers löst die Application im Rahmen des Routings bzw. im Zuge des Dispatchings das Ereignis `dispatch.error` aus. Damit wird signalisiert, dass ein Fehler in der Verarbeitung aufgetreten ist.

6.6 Zusammenfassung

Auf den ersten Blick wirken die Zusammenhänge sehr komplex, die Implementierung des MVC-Pattern im Zend Framework 2 fühlt sich zunächst doch irgendwie “over-engineered” an. Der Hauptgrund für dieses Gefühl ist die Tatsache, dass zum einen die gesamte Verarbeitung in extrem kleine Einzelschritte zerlegt ist, die jeweils über einzelne Klassen abgebildet sind und die, damit sie am Ende auch sinnvoll zusammenspielen, auch noch so zeitlich und inhaltlich “orchestriert” werden müssen, wie es eben erforderlich ist. Zum anderen macht es der exzessive Einsatz von Events und Listnern schwierig, die Abläufe innerhalb der Anwendung und die Zusammenhänge zu verstehen. Die Verwendung etlicher Design-Patterns ist ebenfalls gerade zu Beginn dem Verständnis nicht immer zuträglich, insbesondere nicht, wenn man sie nicht gewohnt ist.

Kann man das nicht alles viel einfacher haben? Muss eine MVC-Implementierung wirklich so komplex sein? Die schnelle, unreflektierte Antwort auf diese Fragen wären: Ja. Nein. Es gibt eine ganze Reihe sog. “Micro MVC Frameworks”, wie etwa [Silex](http://silex.sensiolabs.org/)² oder [MicroMVC](http://micromvc.com/)³, die mit

²<http://silex.sensiolabs.org/>

³<http://micromvc.com/>

deutlich geringerer Komplexität auf den ersten Blick ähnliches zu leisten vermögen, wie die MVC-Implementierung des Zend Framework 2. Allerdings wirklich nur auf den ersten Blick.

Zunächst einmal ist `Zend\Mvc` eigentlich viel mehr als “MVC”. Es ist in Wirklichkeit eine Anwendungsplattform, die es erlaubt, zusätzliche Funktionen in Form von eigenen oder fremden Modulen einfach und effektiv zu integrieren, die Art und Weise der Request-Verarbeitung nahezu nach Belieben zu modifizieren oder ganz und gar umzubauen. Der Ansatz der losen Kopplung der einzelnen Komponenten und Services ermöglicht es, auch den Ansprüchen von Unternehmensanwendungen im Bezug auf Wart-, Erweiterbarkeit und Testbarkeit gerecht zu werden. `Zend\Mvc` schafft ein Ökosystem von Softwarekomponenten und vermag das Abstraktionslevel, auf dem ein Anwendungsentwickler in der Vergangenheit bei der Entwicklung von Webanwendungen mit PHP unterwegs war, auf ein neues, deutliche produktiveres Niveau zu heben. Zumindest tritt es an, genau dies zu erreichen. Ob es wirklich gelingt, gilt es zu beweisen. Ist das Zend Framework 2 das richtige für mein Projekt? Diese Frage wird mit der Version 2 schwieriger zu beantworten, als das noch bei Version 1 der Fall war, meine ich. Die Vorteile von Version 2 zielen schon auf die professionelle Anwendung ab, die nicht immer gegeben ist. Ist das Zend Framework 2 das richtige für meine Unternehmensanwendung im Web? Ja, das würde ich grundsätzlich auf jeden Fall so sagen.

Fassen wir dieses Kapitel und den Ablauf der Request-Verarbeitung noch einmal kurz und knapp zusammen: Zunächst landet der Request durch Einsatz von URL-Rewriting (etwa via “`mod_rewrite`” und der entsprechenden `.htaccess`-Datei) bei der `index.php`. Dort wird das Autoloading konfiguriert und damit sichergestellt, dass sowohl das Zend Framework selbst, aber auch ggf. zusätzlich eingesetzte Bibliotheken ordnungsgemäß funktionieren. Danach wird die `Application` gestartet, die erst einmal dafür sorgt, dass der `ServiceManager` als zentraler “Service-Zugang” erzeugt und mit wichtigen Services ausgestattet wird: Dem `ModuleManager` und dem `EventManager`. Zudem wird der `SharedEventManager` bereitgestellt. Wie im Falle des `ModuleManager` und des `EventManager` heißt bereitstellen häufig, dass zunächst nur die `Factories` bekannt gemacht werden, die jeweils für die spätere Erzeugung der eigentlichen Services herangezogen werden. Warum den Umweg über `Factories` gehen? Zum einen, weil sie es uns ermöglichen, die konkrete Implementierung des jeweiligen Services bei Bedarf auszutauschen. Aber auch, weil im Rahmen der Service-Erzeugung oft nicht nur der Service selbst erzeugt wird, sondern auch `Listener` für die Events in der weiteren Verarbeitung registriert werden. So etwa beim `ModuleManager`. Denn sowohl die `ModuleManagerFactory`, `ApplicationFactory` als auch der `ViewManager` machen im Rahmen der Erzeugung des Services ähnliche Dinge: Sie erzeugen zunächst den eigentlichen Service, dann eine Reihe zusätzlicher (Sub-)Services, auf die der Service später zurückgreift, dann einen oder mehrere `Listener` für Ereignisse des eigenen oder anderer Services. Die `Listener` übernehmen eine bestimmte Aufgabe dann entweder einfach gleich selbst, oder greifen auf die Services zurück.

Dieses Vorgehen führt dazu, dass die Methoden `run`, `loadModules`, `bootstrap` & co. von `ModuleManager`, `Application` & co. sehr schlank daherkommen und eigentlich nichts anderes mehr selbst tun, als die Ereignisse auszulösen. Alles weitere passiert dann über die `Listener` in den Services. Der `ModuleManager` feuert während seiner Ausführung zunächst die Events `loadModules.pre`, `loadModule.resolve`, `loadModule` und `loadModules.post` ab. Im Anschluss dann die `Application` die Ereignisse `bootstrap`, `route`, `dispatch`, zwischendrin der `Controller` mit seinem eigenen

dispatch-Event (nicht zu verwechseln mit dem der Application), dann wieder die Application mit `render`, bevor sich die View mit `renderer` und `response` meldet. Last but not least beendet die Application mit `finish` das Event-Feuerwerk. Im Falle eines Fehlers löst die Application im Rahmen des Routings bzw. im Zuge des Dispatchings das Ereignis `dispatch.error` aus. Damit wird signalisiert, dass ein Problem in der Verarbeitung aufgetreten ist und die entsprechende Fehlerbehandlung greift.

7 Events

Im vorherigen Kapitel haben wir gesehen, wie sehr das ganze Framework auf der Idee von Event-Auslösern, Event-Objekten und Event-Listnern basiert. Dabei wird den entsprechenden Objekten oder “Managern” jeweils ein eigener `EventManager` zur Verfügung gestellt, der sich um die Events des jeweiligen Objektes kümmert, als auch das Hinzufügen und Entfernen von Listnern erlaubt. Weil der `Zend\EventManager` eine so große Wichtigkeit für die Funktion des Frameworks hat, aber auch, weil er bei der Entwicklung der eigenen Anwendung sehr nützlich sein kann, sehen wir ihn uns im Folgenden noch einmal im Detail an.

7.1 Einen Listener registrieren

Der `EventManager` bietet insbesondere zwei Methoden an, die für Listener interessant sind:

```
1 <?php
2 public function attach($event, $callback = null, $priority = 1);
3 public function detach($listener);
```

Listing 7.1

Mit der Methode `attach()` kann ein Listener für ein bestimmtes Event registriert werden, während `detach()` einen Listener entfernt. Die Listener, die für einen Event registriert sind, werden nach der Reihe informiert. Wobei informiert hier meint, dass die jeweils registrierten Callbacks, die, wie man es auch von den nativen PHP-Funktionen gewohnt ist, neben Funktionen, Klassenmethoden, Objektmethoden auch Closures sein dürfen, aufgerufen werden.

Dabei muss zunächst die Bezeichnung des Events angegeben werden, für den der Listener registriert werden soll. Als Konvention wird bei der Bezeichnung eines Events gerne auf die magische Konstante `__FUNCTION__` zurückgegriffen, so dass die Methode, in der das Ereignis ausgelöst wird, gleichzeitig zum Namensgeber für das Ereignis selbst wird. Notwendig ist das aber nicht, der Name ist frei wählbar:

```
1 <?php
2 $greetingService->getEventManager()->attach(
3     'event1',
4     function($e) {
5         // [...]
6     }
7 );
```

Listing 7.2

In diesem Fall wird eine Callback-Funktion für das Ereignis “event1” registriert. Anstelle eines Strings kann auch ein Array mit mehreren Event-Bezeichnungen bei der Registrierung übergeben werden, falls man einen Listener gleich für eine Reihe von Events registrieren will:

```
1 <?php
2 $greetingService->getEventManager()->attach(
3     array('event1', 'event2'),
4     function($e) {
5         // [...]
6     }
7 );
```

Listing 7.3

7.2 Mehrere Listener gleichzeitig registrieren

Es lässt sich aber nicht nur auf einen Schlag ein Listener für eine Reihe von Events registrieren, sondern auch auf einen Schlag eine Reihe von Listnern für ein oder gar gleich mehrere Events. Dazu bedient man sich sog. “Listener Aggregates”. Sie sind insbesondere dann hilfreich, wenn man die Registrierung der einzelnen Listener logisch gruppieren möchte. Dazu stellt man eine eigene Klasse bereit, die das Interface `Zend\EventManager\ListenerAggregateInterface` implementiert und damit über eine Methode `attach()` sowie `detach()` verfügt. Dort werden dann die einzelnen Listener “en bloc” registriert:

```
1  <?php
2  namespace HelloWorld\Event;
3
4  use Zend\EventManager\ListenerAggregateInterface;
5  use Zend\EventManager\EventManagerInterface;
6
7  class MyGetGreetingEventListenerAggregate
8      implements ListenerAggregateInterface
9  {
10     public function attach(EventManagerInterface $eventManager)
11     {
12         $eventManager->attach(
13             'getGreeting',
14             function($e){
15                 // [...]
16             }
17         );
18
19         $eventManager->attach(
20             'refreshGreeting',
21             function($e){
22                 // [...]
23             }
24         );
25     }
26
27     public function detach(EventManagerInterface $events)
28     {
29         // [...]
30     }
31 }
```

Listing 7.4

Das Hinzufügen der Listener wird dann zu einem Einzeiler, weil die Methode `attach()` die Arbeit übernimmt und automatisch aufgerufen wird:

```
1 <?php
2 $greetingService
3     ->getEventManager()
4     ->attach(
5         new \Helloworld\Event\MyGetGreetingEventListenerAggregate()
6     );
```

Listing 7.5

7.3 Einen registrierten Listener entfernen

Über die Methode `detach()` des `EventManager` lässt sich ein bereits registrierter Listener wieder entfernen. Dazu kann man analog zu `attach()` ein `ListenerAggregateInterface` für `detach()` verwenden, oder Listener einzeln entfernen. Dazu übergibt man `detach()` den jeweiligen `CallbackHandler`, den man etwa als Ergebnis des `attach()` zurückbekommen hat:

```
1 <?php
2 $handler = $greetingService->getEventManager()
3     ->attach('getGreeting', function($e){ // [...] });
4
5 $greetingService->getEventManager()->detach($handler);
```

Listing 7.6

7.4 Ein Event auslösen

Der folgende Aufruf reicht aus, um ein Event auszulösen, wenn in der Membervariable `eventManager` des Objekts ein `EventManager` verfügbar ist:

```
1 <?php
2 $this->eventManager->trigger('event1');
```

Listing 7.7

Alle für dieses Event registrierten Listener werden nun aufgerufen. Das ganze passiert natürlich sequentiell und blockierend. Jeder Listener wird einzeln aufgerufen und erst nach vollständiger Verarbeitung wird zum nächsten Listener übergegangen. Die Verarbeitungsreihenfolge wird dabei übrigens durch die Priority festgelegt, die für einen Listener bei `attach()` angegeben werden kann. Oder aber durch die Reihenfolge, in der die Listener hinzugefügt wurden.

Die Methode `trigger()` kann wie oben aufgeführt mit einem String aufgerufen werden, der den Namen des jeweiligen Events repräsentiert, oder aber, wenn ein entsprechendes Event-Objekt übergeben wird:

```
1 <?php
2 $event = new Zend\EventManager\Event();
3 $event->setName('getGreeting');
4 $this->eventManager->trigger($event);
```

Listing 7.7

Der Aufruf von `trigger()` liefert übrigens ein Ergebnis vom Typ `ResponseCollection` zurück, nämlich gesammelt das, was die aufgerufenen Listener zuvor einzeln zurückgegeben haben. Über dessen Methoden `first()` und `last()` lässt sich auf die wichtigsten Rückgabewerte zugreifen, mit `contains($value)` auf einen bestimmten Rückgabewert in der Collection prüfen. Die Ausführungsreihenfolge der Listener ergibt sich dabei wie bereits beschrieben entweder aus der explizit im Rahmen von `attach()` spezifizierten Priorität, oder aber schlicht aus der Reihenfolge, aus der die Listener hinzugefügt wurden. Hier gilt das FIFO-Prinzip: First in, First out. Der als erstes hinzugefügte Listener wird also zuerst ausgeführt.

Und noch eine interessante Sache: Die Abarbeitung der einzelnen Listener lässt sich auch mittendrin unterbrechen. Aber wofür braucht man das? Nehmen wir ein Beispiel: Stell dir vor, du betreibst eine Website, auf der Nutzer eine Rezension zu einem Buch verfassen können. Um sauber zuordnen zu können, fragst du dazu zunächst die ISBN des Buchs ab und sorgst so auch dafür, dass der Buchtitel, Autor usw. nicht mehr manuell vom Nutzer eingegeben werden muss. Dazu greifst du entweder auf Daten zurück, die bereits in deiner Datenbank vorhanden sind (das ist allerdings immer nur dann der Fall, wenn es bereits mindestens eine Rezension zu dem jeweiligen Buch vorher gab), oder du beziehst die Buchdetails von einem entfernten Webservice, etwa dem von Amazon. Wie kannst du die Daten nun also möglichst effizient laden? Du würdest zwei Listener erstellen, die du beide für das Ereignis `onBookDataLoad` deiner Anwendung registrierst, wobei der erste Listener in der Datenbank nach den gewünschten Daten schaut und der zweite im entfernten Webservice. Der zweite Listener wird allerdings nur dann ausgeführt, wenn der erste nicht erfolgreich war. Dazu kann die Auslösung des Ereignisses um ein Callback erweitert werden, in dem auf einen bestimmten Rückgabewert überprüft wird:

```
1 <?php
2 $results = $this->events()->trigger(
3     __FUNCTION__,
4     $this,
5     array(),
6     function ($returnValue) {
7         return ($returnValue instanceof MyModule\Model\Book);
8     }
9 );
```

Listing 7.8

Liefert der erste Listener ein Objekt vom Typ `MyModule\Model\Book` zurück, konnten die Buchdetails also erfolgreich geladen werden, wird die Verarbeitung an dieser Stelle beendet. Alternativ kann auch innerhalb eines Listeners die Verarbeitung beendet werden, wenn dort aktiv die Methode `stopPropagation()` des Event-Objektes aufgerufen wird.

7.5 SharedEventManager

Der `SharedEventManager` ist die Lösung für folgendes Problem: Was macht ein Listener, der sich für ein Event bei einem Event-Auslöser registrieren will, der aber zum Zeitpunkt der Registrierung noch gar nicht existiert? Dieses Problem tritt etwa konkret auf, wenn ein Modul im Rahmen seiner `init()` oder `onBootstrap()` - Methode einen Callback für das Controller-Event “dispatch” registrieren will. Zu diesem Zeitpunkt wurde der Controller mit seinem `EventManager` noch nicht zum Leben erweckt. Es geht schlichtweg so also nicht.

Wird ein neuer `EventManager` über den `ServiceManager` angefordert, so stellt dieser sicher, dass ihm zusätzlich eine Referenz auf den zwischen allen auf diese Art und Weise erzeugten `EventManager`-Instanzen geteilten `SharedEventManager` bekommt. Im `SharedEventManager` können unter Angabe eines “Identifiers” Listener für bestimmte Events registriert werden. Dabei spielt es hier zunächst keine Rolle, ob der `EventManager`, der später den Event auslösen wird, bereits existiert oder nicht. Wichtig ist lediglich die Tatsache, dass man bei der Registrierung des Listeners einen “Identifier” verwendet, für den sich der jeweilige `EventManager` später auch verantwortlich fühlt. Nehmen wir einmal ein konkretes Beispiel, dann wird das Prinzip klar. Wenn die `init()` - Methode der Klasse `Module` unseres `HelloWorld` - Moduls ausgeführt wird, gibt es die `Application` noch nicht. Sie wird erst zum Leben erweckt, wenn das Laden der Module vollständig erledigt ist. Falls wir nun also in der `init()`-Methode einen Listener für das `route` Ereignis der `Application` registrieren wollen, müssen wir dies über den `SharedEventManager` tun, wir haben keine andere Wahl:

```
1  <?php
2  // [...]
3  public function init(ModuleManager $moduleManager)
4  {
5      $moduleManager
6          ->getEventManager()
7          ->attach(
8              ModuleEvent::EVENT_LOAD_MODULES_POST,
9              array($this, 'onModulesPost')
10         );
11
12     $sharedEvents = $moduleManager
13         ->getEventManager()->getSharedManager();
14
15     $sharedEvents->attach(
```

```
16         'application',
17         'route',
18         function($e) {
19             die("Event '{$e->getName()}' wurde ausgelöst!");
20         }
21     );
22 }
```

Listing 7.9

Wir befinden uns in der Datei `Module.php` des Moduls `HelloWorld`. Über den `ModuleManager` kommen wir an dessen `EventManager` und über letzteren wiederum an den `SharedEventManager`, der zwischen allen “EventManager” (die über den `ServiceManager` erzeugt werden) automatisch geteilt wird. Dort registrieren wir nun einen Callback (exemplarisch in Form einer Closure) für das Ereignis `route`, dass später irgendwann einmal die `Application` bzw. dessen `EventManager` auslösen wird. Der Key `application` ist in diesem Fall eine Konvention, ein String, den man kennen muss. In dem Moment, in dem später dann die `Application` mit ihrem eigenen `EventManager`, den wir bis dato noch nicht für die Registrierung des Listeners heranziehen konnten, das Ereignis `route` auslöst, werden auch all die Listener informiert, die sich beim `SharedEventManager` für den Schlüssel `application` und das entsprechende Event registriert haben. Praktisch!

Die folgenden “Identifiers” sind für die einzelnen Framework-Komponenten bereits vorkonfiguriert:

- Für den `ModuleManager`: `module_manager`, `Zend\ModuleManager\ModuleManager`.
- Für die `Application`: `application`, `Zend\Mvc\Application`.
- `AbstractActionController` (die Basisklasse für eigenen “Controller”): `Zend\Stdlib\DispatchableInterface` `Zend\Mvc\Controller\AbstractActionController`, der erste Teil des Namespaces des Controllers (bei `HelloWorld\Controller\IndexController` wäre das etwa `HelloWorld`).
- View: `‘Zend\View\View’`.

Der `SharedEventManager` eignet sich also insbesondere für Situationen, in denen der eigentlich zuständige `EventManager`, den man für die Registrierung eines Listeners heranziehen wollen würde, noch nicht verfügbar ist.

Neben den oben genannten “Identifiers”, die automatisch vom Framework erzeugt werden, können für eigene Zwecke zusätzliche “Identifiers” definiert werden.

7.6 Events in eigenen Klassen verwenden

Der `Zend\EventManager` ermöglicht es einer Klasse, selbst zum Auslöser von Events zu werden und Listener zu verwalten. Seine Funktion ist nicht auf die Klassen und anderen Komponenten des Frameworks begrenzt - ganz im Gegenteil: `Zend\EventManager` eignet sich hervorragend sogar auch als eigenständige Implementierung für die ereignisgesteuerte Verarbeitung.

Dazu muss die eigene Klasse, die Events auslösen soll, lediglich eine Instanz des `Zend\EventManager` in einer Membervariable vorhalten. Nehmen wir noch einmal unseren `GreetingService` von vorhin. Angenommen, wir möchten, immer dann, wenn ein Datum erzeugt wurde, einen Eintrag in unser Logfile schreiben, damit wir jederzeit wissen, wie häufig der Service in einem bestimmten Zeitintervall aufgerufen wurde. Wie könnten wir das realisieren? Schauen wir uns eine Möglichkeit an.

Zunächst einmal legen wir einen weiteren Service in unserem Modul an. Dazu erstellen wir die Datei `src/Helloworld/Service/LoggingService.php` mit folgendem Inhalt:

```
1  <?php
2  namespace Helloworld\Service;
3
4  class LoggingService
5  {
6      public function onGetGreeting()
7      {
8          // Logging-Implementierung
9      }
10 }
```

Listing 7.10

Die konkrete Implementierung des Loggings lassen wir an dieser Stelle zunächst außer Acht, weil es uns ja um das Zusammenspiel verschiedener Services über das Eventsystem geht. Die Methode `onGetGreeting()` wollen wir aufrufen, sobald das Event `getGreeting` des `GreetingService` eintritt.

Zudem stellen wir sicher, dass der Service im System bekannt ist. Dazu fügen wir in der `Module.php` unseres Moduls in der Methode `getServiceConfig()` die entsprechende Klasse als `invokable` hinzu:

```
1  <?php
2  // [...]
3  'invokables' => array(
4      'loggingService' => 'Helloworld\Service\LoggingService'
5  )
6  // [...]
```

Listing 7.11

Der `LoggingService` kann also von nun an über den Key `loggingService` beim `ServiceManager` angefordert werden. Soweit so gut. Nun müssen wir noch dafür sorgen, dass der `GreetingService` ein Ereignis auslösen kann, auf Basis dessen dann die Methode `onGetGreeting` des `LoggingService` ausgeführt wird. Um dem `GreetingService` einen `EventManager` zur Verfügung zu stellen und gleichzeitig die Ausführung der Methode `onGetGreeting` des `LoggingService` zu registrieren, schalten wir dem `GreetingService` eine Factory vor:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class GreetingServiceFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $greetingService = new GreetingService();
12
13         $greetingService->setEventManager(
14             $serviceLocator->get('eventManager')
15         );
16
17         $loggingService = $serviceLocator->get('loggingService');
18
19         $greetingService->getEventManager()
20             ->attach(
21                 'getGreeting',
22                 array($loggingService, 'onGetGreeting')
23             );
24
25         return $greetingService;
26     }
27 }
```

Listing 7.12

Die Factory speichern wir in `src/HelloWorld/Service/GreetingServiceFactory.php`, sie liegt also im gleichen Verzeichnis wie der Service selbst. Die Factory erzeugt zunächst den `GreetingService`, den sie zum Abschluss dann auch zurück gibt. Davor bezieht die Factory für den `GreetingService` noch einen eigenen `EventManager`, so dass der `GreetingService` nun auch Ereignisse auslösen und Listener verwalten kann. Und dann wird noch der `LoggingService` und seine `onGetGreeting()`-Methode für das Ereignis `getGreeting` registriert. Durch den Einsatz einer Closure könnten wir sogar noch dafür sorgen, dass der `$loggingService`, den wir im Beispiel oben direkt anfordern, auch via “Lazy Loading” erst im Moment des tatsächlichen Events angefordert wird:

```
1  <?php
2  namespace Helloworld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class GreetingServiceFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $greetingService = new GreetingService();
12
13         $greetingService->setEventManager(
14             $serviceLocator->get('eventManager')
15         );
16
17         $greetingService->getEventManager()
18             ->attach(
19                 'getGreeting',
20                 function($e) use($serviceLocator) {
21                     $serviceLocator
22                         ->get('loggingService')
23                         ->onGetGreeting($e);
24                 }
25             );
26
27         return $greetingService;
28     }
29 }
```

Listing 7.13

Der Vorteil liegt auf der Hand: Der LoggingService wird tatsächlich erst dann erzeugt, wenn er denn auch genutzt wird. Es könnte ja sein, dass das Ereignis getGreeting gar nicht eintritt.

Bleibt noch, die getServiceConfig() - Methode der Module - Klasse unseres Moduls dahingehend anzupassen, dass der ServiceManager von nun die neue Factory benutzt, um den GreetingService zu erzeugen. Die getServiceConfig() sieht dann also wie folgt aus:

```
1  <?php
2  // [...]
3  public function getServiceConfig()
4  {
5      return array(
6          'factories' => array(
7              'greetingService'
8                  => 'Helloworld\Service\GreetingServiceFactory'
9          ),
10         'invokables' => array(
11             'loggingService'
12                 => 'Helloworld\Service\LoggingService'
13         )
14     );
15 }
```

Listing 7.14

Dem GreetingService bringen wir dann jetzt noch bei, das entsprechende Ereignis auszulösen:

```
1  <?php
2  namespace Helloworld\Service;
3
4  use Zend\EventManager\EventManagerInterface;
5
6  class GreetingService
7  {
8      private $eventManager;
9
10     public function getGreeting()
11     {
12         $this->eventManager->trigger('getGreeting');
13
14         if(date("H") <= 11)
15             return "Good morning, world!";
16         else if (date("H") > 11 && date("H") < 17)
17             return "Hello, world!";
18         else
19             return "Good evening, world!";
20     }
21
22     public function getEventManager()
23     {
```

```
24         return $this->eventManager;
25     }
26
27     public function setEventManager(EventManagerInterface $em)
28     {
29         $this->eventManager = $em;
30     }
31 }
```

Listing 7.15

Und das war's auch schon! Wird die Methode `getGreeting()` aufgerufen, wird das entsprechende Ereignis ausgelöst, für das wir unseren Logger registriert haben.

Wenn wir den `SharedServiceManager` verwenden, können wir die `GreetingServiceFactory` noch weiter vereinfachen:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class GreetingServiceFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $serviceLocator
12             ->get('sharedEventManager')
13             ->attach(
14                 'GreetingService',
15                 'getGreeting',
16                 function($e) use($serviceLocator) {
17                     $serviceLocator
18                         ->get('loggingService')
19                         ->onGetGreeting($e);
20                 }
21             );
22
23         $greetingService = new GreetingService();
24         return $greetingService;
25     }
26 }
```

Listing 7.16

Allerdings muss man dann an geeigneter Stelle - hier exemplarisch einmal direkt im Service vor Auslösen des Ereignisses - dafür sorgen, dass sich der jeweilige EventManager auch für jenen “Identifier” zuständig fühlt. Dazu verwendet man die `addIdentifiers()` - Methode:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  use Zend\EventManager\EventManagerAwareInterface;
5  use Zend\EventManager\EventManagerInterface;
6  use Zend\EventManager\Event;
7
8  class GreetingService implements EventManagerAwareInterface
9  {
10     private $eventManager;
11
12     public function getGreeting()
13     {
14         $this->eventManager->addIdentifiers('GreetingService');
15         $this->eventManager->trigger('getGreeting');
16
17         if(date("H") <= 11)
18             return "Good morning, world!";
19         else if (date("H") > 11 && date("H") < 17)
20             return "Hello, world!";
21         else
22             return "Good evening, world!";
23     }
24
25     public function getEventManager()
26     {
27         return $this->eventManager;
28     }
29
30     public function setEventManager(EventManagerInterface $em)
31     {
32         $this->eventManager = $em;
33     }
34 }
```

Listing 7.17

7.7 Das Event-Objekt

Der Event-Auslöser und die Event-Listener kommunizieren über das Event-Objekt, das vom Event-Auslöser erzeugt und den Event-Listnern beim Aufruf zur Verfügung gestellt wird. Wenn das Ereignis mit Hilfe seines Namens ausgelöst wird

```
1 <?php
2 $this->eventManager->trigger('event1');
```

Listing 7.18

bekommen die Listener ein Objekt vom Typ `Zend\EventManager\Event` übergeben. Es bietet eine interne Datenstruktur zur generischen Übergabe von Parametern, die Informationen zum sog. “Target”, also dem Ort, an dem das Ereignis ausgelöst wurde, sowie die Bezeichnung des Events selbst. Will man den Listnern bestimmte Daten zur Verfügung stellen, kann man dies mit Hilfe von Parametern machen:

```
1 <?php
2 $this->eventManager
3     ->trigger('event1', $this, array("key" => "value"));
```

Listing 7.19

Im Listener kann man dann über die Methode `getParams()` des Event-Objekts auf die Datenstruktur zugreifen. Mit Hilfe von `Zend\EventManager\Event` lässt sich also im Grunde bereits alles machen, was man so braucht. Möchte man aber statt der generischen Datenstruktur mit konkreten Bezeichnungen und Membervariablen arbeiten, auf die über Getter und Setter zugegriffen werden kann, so hat man als Anwendungsentwickler die Möglichkeit, eine eigene Event-Klasse anzugeben, die bei Bedarf “on-the-fly” instanziiert und gefüllt wird. Das Zend Framework 2 macht selbst regen Gebrauch von diesem Mechanismus, gibt es doch ein `ModuleEvent`, ein `ViewEvent` und auch ein `MvcEvent`, dass etwa direkten Zugriff auf die folgenden Objekte ermöglicht:

```
1 <?php
2 protected $application;
3 protected $request;
4 protected $response;
5 protected $result;
6 protected $router;
7 protected $routeMatch;
8
9 // [...]
```

Listing 7.20

Für eine eigene Event-Klasse kann man am besten von `Zend\EventManager\Event` ableiten:

```
1  <?php
2  namespace HelloWorld\Event;
3
4  use Zend\EventManager\Event;
5
6  class MyEvent extends Event
7  {
8      private $myObject;
9
10     public function setMyObject($myObject)
11     {
12         $this->myObject = $myObject;
13     }
14
15     public function getMyObject()
16     {
17         return $this->myObject;
18     }
19 }
```

Listing 7.21

Um die Event-Klasse zu registrieren, kann man nun entweder diese Information vorher bekannt machen und dann das Ereignis wieder durch den Ereignisnamen aufrufen

```
1  <?php
2  $this->eventManager->setEventClass('HelloWorld\Event\MyEvent');
3  $this->eventManager->trigger('getGreeting');
```

Listing 7.22

oder man übergibt das entsprechende Objekt im Rahmen von `trigger()`:

```
1  <?php
2  $event = new \HelloWorld\Event\MyEvent();
3  $event->setName('getGreeting');
4  $this->eventManager->trigger($event);
```

Listing 7.23

Auf diese Art und Weise könnte man auch vollständig eindeutig Event-Klassen schreiben, bei denen auch der Name bereits vordefiniert ist:

```
1  <?php
2  namespace HelloWorld\Event;
3
4  use Zend\EventManager\Event;
5
6  class MyGetGreetingEvent extends Event
7  {
8      private $myObject;
9
10     public function __construct()
11     {
12         parent::__construct();
13         $this->setName('getGreeting');
14     }
15
16     public function setMyObject($myObject)
17     {
18         $this->myObject = $myObject;
19     }
20
21     public function getMyObject()
22     {
23         return $this->myObject;
24     }
25 }
```

Listing 7.24

Die Auslösung dieses Events wäre dann noch weniger fehleranfällig und der Code noch kompakter:

```
1  <?php
2  $event = new \HelloWorld\Event\MyGetGreetingEvent();
3  $this->eventManager->trigger($event);
```

Listing 7.25

8 Module

Neben dem `EventManager` spielt das Konzept der “Module” eine entscheidende Rolle in Zend Framework 2. Dem `ModuleManager` kommt in der Request-Verarbeitung in diesem Rahmen eine große Bedeutung zu, sorgt er doch dafür, dass die aktivierten Module überhaupt berücksichtigt und geladen werden, die Anwendung also Funktionalität bekommt.

Wichtig ist die Tatsache, dass die Module der Anwendung zur Laufzeit keine “in sich geschlossenen Einheiten” bilden. Das Gegenteil ist der Fall: Die Funktionen der einzelnen Module verschmelzen im Rahmen des Ladens der Module zur “Gesamtfunktionalität” der Anwendung. Wir haben gesehen, dass der `ModuleManager` die Konfigurationen aller Module vereint in ein anwendungsweites Konfigurationsobjekt. Diese Tatsache hat einige Konsequenzen, derer man sich als Anwendungsentwickler bewusst sein sollte. Alle “Services”, die ein Modul zur Verfügung stellt, sind anwendungsweit verfügbar, also etwa “Controller Plugins” oder “View Helper”. Auch z.B. die Controller des einen Moduls stehen grundsätzlich den anderen Modulen zur Verfügung; es gibt keinen separaten `ControllerLoader` pro Modul.

8.1 Das Modul “Application”

Startet man die Entwicklung der eigenen Anwendung auf Basis der `ZendSkeletonApplication`, so stolpert man recht schnell über das Modul `Application`. Ist man sich der oben beschriebenen Tatsache bewusst, dass die Funktionalitäten aller Module gemeinsam die Gesamtfunktionalität der Anwendung ergeben und riskiert man mal einen Blick in das, was da über dieses Modul bereitgestellt wird, so wird schnell der Sinn und Zweck dieses “Standard-Moduls” klar: Es konfiguriert eine Reihe von Services und erzeugt einige Basisfunktionen, die jede Anwendung benötigt und dessen eigene Implementierung man sich dann also schenken kann. Neben der Tatsache, dass die `ZendSkeletonApplication` einen exemplarischen Controller für die “Startseite” der Anwendung samt dessen Routen-Konfiguration mitbringt, kommen über die `module.config.php` der `Application` auch eine Reihe von Einstellungen für den “View Layer”, die von der `ViewManagerFactory` oder dem `ViewManager` ausgelesen und entweder selbst verwertet oder an andere Objekte durchgereicht werden:

```
1  <?php
2  // [...]
3  'view_manager' => array(
4      'display_not_found_reason' => true,
5      'display_exceptions' => true,
6      'doctype' => 'HTML5',
7      'not_found_template' => 'error/404',
8      'exception_template' => 'error/index',
9      'template_map' => array(
10         'layout/layout'
11             => __DIR__ . '/../view/layout/layout.phtml',
12         'application/index/index'
13             => __DIR__ . '/../view/application/index/index.phtml',
14         'error/404'
15             => __DIR__ . '/../view/error/404.phtml',
16         'error/index'
17             => __DIR__ . '/../view/error/index.phtml',
18     )
```

Listing 8.1

- Über `display_not_found_reason` wird die `RouteNotFoundStrategy` (die standardmäßige Art und Weise, wie 404-Fehler behandelt werden) angewiesen, den Grund für die Tatsache, dass eine URL zu einem 404-Fehler geführt hat, zur weiteren Darstellung bereitzustellen. Wie wir in den vorangegangenen Kapiteln gesehen haben, liegt einem jeden Ergebnis einer Requestverarbeitung das “View Model” zu Grunde, dass sowohl die anzuzeigenden Nutzdaten, als das dazu heranzuziehende (HTML-)Template hält. Wenn ein 404-Fehler auftritt, gibt es allerdings in aller Regel kein von einem Controller erzeugtes “View Model”, eben weil es keinen zuständigen Controller gab. Die `RouteNotFoundStrategy` sorgt dann dafür, dass für das weitere Rendering ein “View Model” erzeugt wird, so dass das überhaupt stattfinden kann. Ein passendes Template bringt die `ZendSkeletonApplication` ebenfalls in `view/error/404.phtml` mit. Dort kann über `$this->reason` auf die Problemursache zugegriffen werden.
- Mit `not_found_template` wird explizit das Template bestimmt, das in 404-Situationen herangezogen werden soll. Standardmäßig sucht das Framework nach dem Template `404.phtml`. Es würde also ausreichen, wenn man direkt in das `view`-Verzeichnis eines der Module ein entsprechendes Template `404.phtml` anlegen würde. Legt man in mehr als ein Modul eine Datei `404.phtml`, würde übrigens die des zuletzt aktivierten Moduls verwendet. Wir erinnern uns: Die Konfigurationen der einzelnen Module werden beim Laden durch den `ModuleManager` zusammengefügt. In der `ZendSkeletonApplication` wird das Template `error/404` für 404-Situationen bestimmt, dass wiederum über die nachfolgende `template_map` - Konfiguration auf ein physisches File zeigt:

```
1 'error/404' => __DIR__ . '/../view/error/404.phtml'
```

Listing 8.2

- `display_exceptions` macht ähnliches wie `display_not_found_reason`: Hier wird analog die `ExceptionHandler` angewiesen, `$this->display_exceptions` im entsprechenden “View Model” und damit im definierten Template verfügbar zu machen. So kann dort entschieden werden, ob weitere Details der Exception angezeigt werden sollen oder nicht.
- Mit `exception_template` wird explizit das Template bestimmt, das in Exception-Situationen, sprich immer dann, wenn irgendwo in der Verarbeitung eine Exception “fliegt”, die nicht auch irgendwo vom Anwendungsentwickler aufgefangen und verarbeitet wird, herangezogen wird. Standardmäßig sucht das Framework nach dem Template `error.phtml`. Es würde also ausreichen, wenn man direkt in das `view`-Verzeichnis eines der Module ein entsprechendes Template `error.phtml` legen würde. In der `ZendSkeletonApplication` wird das Template `error/index` für Exception-Situationen bestimmt, dass wiederum über die nachfolgende `template_map`-Konfiguration auf ein physisches File zeigt:

```
1 'error/index' => __DIR__ . '/../view/error/index.phtml'
```

Listing 8.3

- Über `doctype` wird der `Doctype-View-Helper` konfiguriert, den wir uns später noch genauer ansehen werden. Über ihn kann dann die `Doctype-Deklaration` in einem Template oder Layout (auch dazu später mehr) ausgegeben werden, ohne sie manuell hinzufügen zu müssen. Der hinterlegte Wert wird von der `ViewHelperManagerFactory` verwendet, um den “View Helper” `doctype` mit der entsprechenden Konfiguration zu versehen.
- Dann wird über den Schlüssel `layout` das Template definiert, das als “visueller Rahmen” fungiert. Standardmäßig erwartet das Framework das Layout-Template unter dem Schlüssel `layout/layout` in der Template-Map, bzw. in einem entsprechenden File im Dateisystem. Über den Key `layout` ließe sich bei Bedarf ein abweichendes Template angeben:

```
1 'layout' => 'myLayout'
```

Listing 8.4

Das Modul “Application” ist also grundsätzlich erst einmal nichts besonderes, nimmt dem Anwendungsentwickler aber initialen Konfigurationsaufwand ab. Ich empfehle, dass Modul “Application” auch bewusst weiterzuentwickeln und dort sämtliche Definitionen und Konfigurationen abzulegen, die alle oder der Großteil der Module gemeinsam haben. Theoretisch könnte man eine Anwendung auch ganz so gestalten, dass sie neben dem Modul “Application” keine weiteren Module mehr enthält und sämtliche Funktionen direkt dort realisiert sind. In einigen Fällen ist dies sicherlich sogar ein probates Vorgehen.

8.2 Modulabhängiges Verhalten

Aufgrund der Tatsache, dass es nach dem Laden der Module im Grunde keine Module mehr in der laufenden Anwendung gibt, sondern nur noch die Anwendung selbst mit all ihren Funktionen und Konfigurationen, die sich aus denen der einzelnen Modulen zusammengefügt haben, bietet sich im Grunde keine Möglichkeit, zu einem späteren Zeitpunkt ein modulabhängiges Verhalten zu konfigurieren, schlichtweg, weil keine Information zum “gerade aktiven Modul” zur Verfügung steht. Will man aber nun eine bestimmte Aktion ausführen, etwa wenn der Controller eines bestimmten Moduls ausgeführt wird, so muss man auf den `SharedEventManager` und einen Kniff zurückgreifen:

```
1  <?php
2  namespace HelloWorld;
3
4  use Zend\ModuleManager\ModuleManager;
5
6  class Module
7  {
8      public function init(ModuleManager $moduleManager)
9      {
10         $sharedEvents = $moduleManager->getEventManager()
11             ->getSharedManager();
12
13         $sharedEvents->attach(
14             __NAMESPACE__,
15             'dispatch',
16             function($e) {
17                 $controller = $e->getTarget();
18                 $controller->layout('layout/helloWorldLayout');
19             },
20             100
21         );
22     }
23 }
```

Listing 8.5

In diesem Beispiel konfigurieren wir ein abweichendes Layout für alle Seiten, die über eine Controller-Aktion unseres HelloWorld-Modules erzeugt werden. Dazu binden wir eine Callback-Funktion an das Event `dispatch`, allerdings nur für die Controller, die sich für den “Identifizier” `HelloWorld` (dieser Wert entspricht der magischen Konstante `__NAMESPACE__` in diesem Fall) verantwortlich fühlen - also für alle Controller des HelloWorld-Moduls. Wie genau funktioniert das? Wir haben hier wieder die besondere Situation, dass wir einen Listener (in diesem Fall in Form einer

Callback-Funktion) bei einem “EventManager” registrieren wollen, den es zum Zeitpunkt der Registrierung noch gar nicht gibt. Ein Controller bekommt ja erst zum Zeitpunkt seiner Erzeugung seinen eigenen “Event Manager” zugeteilt. Das heißt, wir müssen dazu den `SharedEventManager` nutzen (siehe auch vorheriges Kapitel). Und jetzt kommt der Kniff: Ein Controller im Zend Framework 2 ist immer dann, wenn er vom `AbstractActionController` ableitet, dahingehend konfiguriert, dass er bei der Benachrichtigung seiner Listener den `SharedEventManager` unter anderem auch unter Angabe des Controller-Namespace (also in diesem Fall “Helloworld”) konsultiert. Und genau dafür haben wir in der `init()`-Methode oben den Callback implementiert, der dann aufgerufen wird, über das `MvcEvent`-Objekt den ermittelten Controller bezieht und mit Hilfe des “Controller Plugins” `layout` ein anderes Layout-Template bestimmt. Das Template selbst muss es natürlich geben bzw. es muss zusätzlich über die Template-Map verfügbar gemacht werden. Ansonsten führt dies zu einem Fehler.

Noch einmal anders ausgedrückt: Wenn man im `SharedEventManager` Listener für die Events eines Identifiers registriert, die dem Namespace eines Moduls entsprechen, werden diese von den Controllern des jeweiligen Moduls berücksichtigt.

8.3 Ein Fremdmodul installieren

Einer der großen Errungenschaften der Version 2 ist die Tatsache, dass die eigene Anwendung auf einfache Art und Weise um Funktionen erweitert werden kann, ohne, dass man sie selbst programmieren müsste. Für die Installation eines Fremdmoduls sind ein paar grundsätzliche Arbeitsschritte erforderlich und je nach Modul ggf. weitere Handgriffe.

Quellen für Fremdmodule

Zwei gute Quellen für den Bezug von Module sind die [offizielle ZF-Module-Seite](http://modules.zendframework.com/)¹ und [GitHub](https://github.com)², wobei dort insbesondere das [ZF-Commons-Repository](https://github.com/ZF-Commons)³ zu erwähnen ist. Über die Suchfunktion bei Github lassen sich aber schnell weitere Repositories finden. An dieser Stelle ein kurzer Hinweis: Nicht alle verfügbaren Module verfügen über die Qualität, die man ggf. selbst als den Minimumstandard für seinen Code ansetzt. Viele Module, gerade die mit sehr kleinen Versionsnummern, enthalten noch zahlreiche Bugs und Sicherheitslücken. Durch die Tatsache, dass die Funktionen und Konfigurationen aller Module durch den `ModuleManager` im Rahmen des Ladens der Module vereinigt werden, können zuvor registrierte Services und ähnliches auf einmal nicht mehr verfügbar sein, etwa weil sie unbeabsichtigt mit anderen Implementierungen überschrieben wurden. Es gilt also, Fremdmodule mit wachsamem Auge einzusetzen und bewusste Entscheidungen für die Nutzung oder Nichtnutzung eines Moduls zu treffen.

¹<http://modules.zendframework.com/>

²<https://github.com>

³<https://github.com/ZF-Commons>

Installation eines Fremdmoduls

Es gibt ein Vielzahl von Möglichkeiten, zusätzliche Module in der eigenen Anwendung verfügbar zu machen. Die einfachste Art und Weise ist der manuelle Download des Module-Codes und das Kopieren der Sourcefiles in die eigene Anwendung. So kann man etwa das [Modul ZfcTwig herunterladen](#)⁴ und den Inhalt des Archivs in den `module`-Folder der eigenen Anwendung kopieren (z.B. in ein Verzeichnis `“ZfcTwig”`). Denkt man dann noch daran, das Modul in der `application.config.php` zu aktivieren (im Abschnitt `modules` den Wert `“ZfcTwig”` hinzufügen), ist das Modul soweit lauffähig. Allerdings handelt es sich bei `ZfcTwig` um ein Modul, dass seinerseits noch die PHP-Bibliothek `“Twig”`⁵ benötigt, damit es tatsächlich einsatzbereit ist. `“Twig”` ist eine [Template-Enginge](#)⁶ von den Machern des Symfony-Frameworks, einem der großen Rivalen des Zend Frameworks. `“Twig”` hat dem alten Platzhirschen `“Smarty”`⁷ mittlerweile so ein wenig den Rang abgelaufen und wird gerne und bereits oft eingesetzt. Man kann sich übrigens zurecht fragen, warum man für die Entwicklung von Templates in PHP eigentlich überhaupt eine weitere Template-Sprache wie `“Twig”` benötigt, wenn doch PHP selbst eine Template-Sprache ist. Schließlich bestehen die typischen `“phtml”`-Files von Zend Framework ja auch nur aus HTML, PHP-Code und bei Bedarf einigen `“View Helpern”` und das Zend Framework 2 bringt, ebenso wie Version 1, von Haus aus auch keine andere Templating-Engine mit. Diese Fragen sind alle sehr richtig und sehr valide. Die kurze Antwort darauf lautet: Nein, eigentlich brauchen wir kein zusätzliches Templating-System, dessen Syntax wir zusätzlich lernen und dessen Ecken und Kanten wir kennen müssen. Alles ist gut so, wie es ist. Es gibt allerdings Anwendungsfälle, in denen kann eine zusätzliche Template-Engine hilfreich sein, etwa dann, wenn man verhindern möchte, dass in den Templates `“Pfusch”` betrieben werden kann. Ein rein PHP-basiertes Template bietet ja alle Möglichkeiten von PHP, etwa den Zugriff auf alle Funktionen des Sprachkerns. Wenn wir das verhindern und die Möglichkeiten in den Templates bewusst auf ein definiertes Set von Funktionen beschränken wollen, kann eine Template-Engine hilfreich sein - auch jenseits des `“syntactic sugar”`, dem anderen Hauptgrund für ihren Einsatz.

Aber zurück zum eigentlichen Problem der Abhängigkeit des Modules `ZfcTwig` von der `“Twig”`-Bibliothek: Das `ZfcTwig`-Modul fungiert also als eine Art `“Glue Code”` und sorgt dafür, dass die Funktionen von `“Twig”` im Kontext einer Zend Framework 2 Anwendung genutzt werden können. Wir können nun also als nächstes `“Twig”` herunterladen, im Verzeichnis `vendor` ablegen und an den geeigneten Stellen das Autoloading der `“Twig”`-Klassen konfigurieren. Dann wird es funktionieren.

Weil dieser Prozess aber fehleranfällig und aufwändig ist, empfiehlt sich auch für die Installation von Fremdmodulen wann immer möglich der Einsatz von `“Composer”`⁸, so wie bereits zuvor bei der Installation der `ZendSkeletonApplication` selbst. Machen wir es nämlich so, kümmert sich `“Composer”` nicht nur um den Download und die Bereitsstellung des `ZfcTwig`-Moduls, sondern auch um dessen Abhängigkeiten, also die `“Twig”`-Bibliothek. Dazu erweitern wir die `composer.json` unserer Anwendung im Abschnitt `require`:

⁴<https://github.com/ZF-Commons/ZfcTwig/tarball/master>

⁵<http://twig.sensiolabs.org/>

⁶http://de.wikipedia.org/wiki/Template_Engine

⁷<http://www.smarty.net/>

⁸getcomposer.org

```
1  "require": {  
2      "zf-commons/zfc-twig": "dev-master"  
3  }
```

Listing 8.6

und führen “Composer” im Projektverzeichnis erneut aus

```
1  $ php composer.phar update
```

“Composer” lädt nun die notwendigen Bibliotheken herunter und richtet auch das Autoloading für uns ein. Schaut man nun allerdings in das `module`-Verzeichnis der eigenen Anwendung, so stellt man fest, dass dort kein Modul hinzugekommen ist. “Composer” legt stattdessen alle über ihn geladenen Bibliotheken standardmäßig im Verzeichnis `vendor` ab. In der `application.config.php` steuert der Abschnitt

```
1  <?php  
2  // [...]   
3  'module_paths' => array(  
4      './module',  
5      './vendor',  
6  ),
```

Listing 8.7

die Pfade, in denen die Anwendung installierte Module erwartet. `vendor` ist also vollkommen okay und selbst ganz andere Pfade können an dieser Stelle konfiguriert werden.

Öffnet man jetzt `/sayhello` so sieht man ... Buchstabensalat. Die Templates nämlich, die wir derzeit noch in unserer `HelloWorld`-Anwendung im Einsatz haben, basieren auf PHP und sind damit für “Twig” unverständlich. “Twig” benötigt ja “Twig”-konformes Template-Markup. Glücklicherweise liefert `ZfcTwig` alternative, “Twig”-kompatible Templates für die `ZendSkeletonApplication`-Beispielseiten mit, die man anstelle der “normalen” Templates des Moduls `application` verwenden kann. Nach Installation von `ZfcUser` via “Composer” findet man jene unter `vendor/zf-commons/zfc-twig/examples`. Mit diesen Dateien überschreibt man die jetzigen Templates in `module/application/view`. Dann haben wir schon einmal das Layout und die Templates für die Fehlerseiten “Twig”-kompatibel gemacht. Gleiches muss man jetzt natürlich auch noch mit den selbstgeschriebenen Templates tun. Danach kann man `/sayhello` wie gewohnt aufrufen. Die Templates werden nun durch “Twig” verarbeitet.

8.4 Ein Fremdmodul konfigurieren

Andere Fremdmodule haben mehr die Gestalt von “MVC”-Modulen, bringen Controller, Views, Routen & Co. mit. Ein gutes Beispiel für so ein Modul ist [ZfcUser](https://github.com/ZF-Commons/ZfcUser)⁹, das vollständig die Funktionen einer Nutzerregistrierung samt Einloggen/Ausloggen - Flows abbildet. Wir werden uns dieses Modul zu einem späteren Zeitpunkt auch noch einmal im Detail ansehen.

Stellen wir uns der Einfachheit für einen Moment lang vor, dass das `Helloworld`-Modul der letzten Kapitel ein Fremdmodul wäre, dass wir über “Composer” installiert hätten. Mit dem `Helloworld`-Modul wäre jetzt also die URL `/sayhello` in unser System gekommen. Was aber, wenn wir die entsprechende Seite eigentlich unter der URL `/welcome` bereitstellen wollen?

Natürlich könnten wir jetzt die `module.config.php` des Moduls öffnen und unsere Anpassungen dort direkt vornehmen. Das würde aber bedeuten, dass wir die Codebase von `Helloworld` “geforked” hätten und damit Verbesserungen, die an `Helloworld` durch den ursprünglichen Autor (wir tun ja grad so, als wäre es ein Fremdmodul) vorgenommen werden, nicht mehr einfach so in unser System holen könnten. Bei jedem Update würden wir unsere Modifikationen überschreiben, müssten sie ggf. jedes Mal wieder manuell nachpflegen. Alles nicht optimal.

Stattdessen werden wir unsere Anpassungen in einem eigenen Modul unterbringen und damit von der Codebase von `Helloworld` separieren. Auf diesem Wege behält `Helloworld` seine ursprüngliche Gestalt und könnte jederzeit gefahrlos aktualisiert werden, ohne dass unsere Änderungen verloren gehen.

URLs ändern

Um die URL `/sayhello` in `/welcome` zu ändern legen wir ein neues Unterverzeichnis und damit ein neues Modul im Verzeichnis `module` an und nennen es `HelloworldMod`. So machen wir deutlich, dass es sich hier um eigene Modifikationen eines anderen Moduls handelt. Dort benötigen wir nur die `Module.php`, in dessen `getConfig()`-Methode wir die Routendefinition für `sayhello` aufgreifen und mit `/welcome` überschreiben:

```
1  <?php
2  namespace HelloWorldMod;
3
4  class Module
5  {
6      public function getConfig()
7      {
8          return array (
9              'router' => array(
10                 'routes' => array(
```

⁹<https://github.com/ZF-Commons/ZfcUser>

```

11         'sayhello' => array(
12             'type' => 'Zend\Mvc\Router\Http\Literal',
13             'options' => array(
14                 'route' => '/welcome'
15             )
16         )
17     )
18 )
19 );
20 }
21
22 // [...]
23 }

```

Listing 8.8

Nun müssen wir noch dafür sorgen, dass das Modul aktiviert wird und erweitern dazu entsprechend die `application.config.php` im Abschnitt `modules`:

```

1  <?php
2  return array(
3      'modules' => array(
4          'Application',
5          'Helloworld',
6          'HelloworldMod'
7      ),
8      'module_listener_options' => array(
9          'config_glob_paths' => array(
10             'config/autoload/{,*.}{global,local}.php',
11         ),
12         'module_paths' => array(
13             './module',
14             './vendor',
15         ),
16     ),
17 );

```

Listing 8.9

Wenn wir jetzt die URL `/sayhello` aufrufen, bekommen wir wie erwartet einen 404-Fehler. `/welcome` hingegen liefert nun die gewohnte Seite.

Wir machen uns hier zwei Mechanismen zu Nutze: Erstens wird die Konfiguration aller Module im Rahmen ihres Ladens vereinigt, so dass wir in einem der Module sozusagen auch Konfigurationen

“für andere Module” machen können. Zweitens wird die Konfiguration der Module einzeln nacheinander eingelesen. In unserem Fall also zunächst die der Application, dann die von `HelloWorld` und dann die Konfiguration des `HelloWorldMod`-Moduls. Auf diese Weise können Konfigurationen eines vorherigen Moduls von den nachfolgenden überschrieben werden, so wie wir dies für die Route mit der Bezeichnung `sayhello` tun, die zunächst von `HelloWorld` definiert und dann von `HelloWorldMod` im Nachgang in seiner Option `route` zu `/welcome` “umkonfiguriert” wird.

Views ändern

Auch die Darstellung von Fremdmodulen lässt sich anpassen. Dies ist in aller Regel wichtiger und häufiger erforderlich, als die Anpassung von URLs. Erweitern wird dazu die `Module`-Klasse von `HelloWorldMod` um den Abschnitt `view_manager` und biegen dort das Template der Action `index` des Controllers `index` des Moduls `HelloWorld` auf das entsprechend modifizierte Template im `HelloWorldMod`-Modul um:

```
1  <?php
2  namespace HelloWorldMod;
3
4  class Module
5  {
6      public function getConfig()
7      {
8          return array (
9              'router' => array(
10                 'routes' => array(
11                     'sayhello' => array(
12                         'type' => 'Zend\Mvc\Router\Http\Literal',
13                         'options' => array(
14                             'route' => '/welcome'
15                         )
16                     )
17                 )
18             ),
19             'view_manager' => array(
20                 'template_map' => array(
21                     'helloworld/index/index'
22                         => __DIR__ .
23                             '/view/helloworld-mod/index/index.phtml'
24                 )
25             )
26         );
```

```
27         }  
28     }
```

Listing 8.10

Rufen wir nun `/welcome` auf, so sehen wir die erzeugte Seite unter Nutzung des neuen Templates.

9 Controller

9.1 Konzept & Funktionsweise

Ein Controller hat die Aufgabe, eine Interaktion mit der Benutzeroberfläche, in unserem Fall also einer Website bzw. im weiteren Sinne mit dem Browser selbst, zu verarbeiten.

Das MVC-Pattern (das “C” steht ja hier für “Controller”) ist übrigens schon recht alt. Es wurde erstmals Ende der 70er Jahre für die Realisierung von Benutzeroberflächen eingesetzt, als die Programmiersprache [Smalltalk](http://de.wikipedia.org/wiki/Smalltalk)¹ noch populär war. Die Benutzeroberflächen sowie die Verarbeitung der Interaktion war seinerzeit allerdings in aller Regel auf ein abgeschlossenes System beschränkt. Im Web ist dies anders: Die Oberfläche manifestiert sich im Browser, also auf dem Client, während die Verarbeitung auf dem Server stattfindet. Der Browser überträgt für uns die Information darüber, was mit ihm selbst bzw. mit der jeweiligen Website, die er gerade anzeigt, getan wird, auf welche Art und Weise der Nutzer mit ihr interagiert. Klickt der Nutzer auf einen Link, überträgt der Browser die Tatsache, dass der Anwender gerade eine andere Seite angefordert hat, an den Server. Auf dem Server ermittelt das Framework jetzt, welcher Controller im System für die Verarbeitung dieser Interaktion vorgesehen ist (Routing) und übergibt diesem die weitere Verantwortung für die Erzeugung der Response, die später im Browser des Anwenders für eine Veränderung der Darstellung sorgt. Entweder wird eine komplett neue Seite geladen, oder, falls AJAX verwendet wird, werden nur bestimmte Bereiche der bereits angezeigten Seite aktualisiert.

Üblicherweise legt man für jeden “Seitentyp” einen eigenen Controller im System an. In einem Online-Shop gäbe es also zum Beispiel einen “IndexController” für die Homepage, einen “CategoryController” für die Listendarstellung von Angeboten in einer bestimmten Kategorie und einen “ItemController” für die Darstellung einer Angebotsdetailseite. Ein Controller wird im Zend Framework weiter in sog. “Actions” unterteilt. Die eigentliche Verarbeitung findet also nicht im Controller selbst, sondern in dessen Actions statt. Actions sind dabei öffentliche Methoden der Controller-Klasse. Neben den oben genannten Controllern bietet ein Online-Shop in aller Regel auch einen Warenkorb und das System hätte daher vermutlich zudem einen “CartController”. Um die üblichen Interaktionen mit einem Warenkorb abzubilden, wäre der “CartController” mit einer Reihe von Actions ausgestattet, darunter etwa die “showAction”, “addAction”, “removeAction”, “removeAllAction” und so weiter.

Ein sinnvolles Vorgehen ist es, den Code innerhalb von Controllern bzw. Actions schlank zu halten. Ein Controller sollte im Idealfall nur die Interaktion registrieren, auswerten und dann entscheiden, welche Hebel in Bewegung zu versetzen sind, um das gewünschte Ergebnis zu erzeugen. In aller Regel bedient er sich dazu dann Services, die etwa den Zugriff auf Datenbanken, Sessions oder sonstigen Informationen und Funktionen ermöglichen.

¹<http://de.wikipedia.org/wiki/Smalltalk>

9.2 Controller Plugins

Zusätzlich bedienen sich Controller oft sog. “Controller Plugins”, die interaktionsrelevanten Code kapseln, der häufiger benötigt wird und auf diese Art und Weise in unterschiedlichen Controllern wiederverwendet werden kann.

Das Framework kommt mit einer ganzen Reihe von Plugins, die sofort in eigenen Controllern eingesetzt werden können. Alle Plugins erben von derselben Basisklasse, die sicherstellt, dass das jeweilige Plugin immer auch Zugriff auf den Controller hat, in dem es gerade zum Einsatz kommt. Das ist in vielen Situationen praktisch, in manchen sogar unabdingbar, wie wir im Folgenden sehen werden.

Redirect

Das Redirect-Plugin ermöglicht es, einen Client auf eine andere URL umzuleiten. Dieser Redirect geschieht dabei nicht “Framework-intern”, sondern durch Rückmeldung an den Client, der dann aktiv die neue URL aufruft. Realisiert wird dies dadurch, dass das Framework eine Response mit einem 302 [HTTP-Statuscode](http://de.wikipedia.org/wiki/HTTP-Statuscode)² sendet und der Client, in aller Regel also der Browser des Nutzers, einen neuen Request auf die angegebene URL durchführt.

Dazu kann direkt eine Ziel-URL eingetragen werden:

```
1 <?php
2 $this->redirect()->toUrl('http://www.meinedomain.de/zielseite');
```

Listing 9.1

Oder aber eine URL auf Basis der im System verfügbaren Routen erzeugt werden. Wollen wir also etwa auf die URL `http://localhost:8080/sayhello` weiterleiten, so können wir alternativ die URL auf Basis der Routenkonfiguration generieren lassen:

```
1 <?php
2 $this->redirect()->toRoute('sayhello');
```

Listing 9.2

Hierbei ist `sayhello` der Name der Route, den wir in unserer Module-Konfigurationsdatei `module.config.php` angegeben haben:

²<http://de.wikipedia.org/wiki/HTTP-Statuscode>


```
1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'sayhello' => array(
6              'type' => 'Zend\Mvc\Router\Http\Literal',
7              'options' => array(
8                  'route' => '/sayhello',
9                  'defaults' => array(
10                     'controller' => 'Helloworld\Controller\Index',
11                     'action' => 'index',
12                 )
13             )
14         )
15     )
16 )
```

Listing 9.3

Was aber, wenn die URL kein statischer String ist wie in dem Beispiel, sondern sich dynamisch zusammensetzt, etwa wie bei `http://www.zalando.de/nike-performance-free-run-3-laufschuh-black-relfectin`. Auch hier gibt es eine Lösung: Zusätzlich zum Namen der Route können als zusätzliche Parameter die dynamischen Bestandteile, aus denen sich die URL zusammensetzt, übergeben werden. Wir schauen in einem der späteren Kapitel noch einmal im Detail auf den Mechanismus des Routings und kehren dann auch noch einmal inhaltlich zum Redirect-Plugin zurück. Dann wird klar, wie genau für eine Ziel-URL die dynamischen Bestandteile erzeugt werden.

Das Redirect-Plugin macht sich die Tatsache zu nutze, dass wenn ein Controller ein Objekt vom Typ `Response` zurückgibt, die sonst übliche Weiterverarbeitung übergangen wird. Es findet in diesem Fall sinnvoller Weise also kein Rendering der View mehr statt.

Damit das Redirect-Plugin ordnungsgemäß funktioniert, ist es erforderlich, dass der Controller, in dem es eingesetzt wird, über das `MvcEvent`-Objekt verfügt, wird darüber doch der Router bezogen, der notwendig ist, um eine URL auf Basis eines Routen-Namens zu erzeugen. Üblicherweise leitet man eigene Controller immer von der Framework-Basisklasse `AbstractActionController` ab, die bereits eine ganze Reihe von Controller-relevanten Interfaces implementiert und so etwa sicherstellt, dass das `MvcEvent` verfügbar ist:

```
1  <?php
2  namespace Zend\Mvc\Controller;
3
4  // use [...];
5
6  abstract class AbstractActionController implements
7      Dispatchable,
8      EventManagerAwareInterface,
9      InjectApplicationEventInterface,
10     ServiceLocatorAwareInterface
11  {
12      // [...]
13  }
```

Listing 9.4

Der `AbstractActionController` des Frameworks implementiert das `InjectApplicationEventInterface` und ermöglicht es damit dem `ControllerLoader`, dem Service, der nach dem Routing dafür zuständig ist, den zur Route passenden Controller zu instanzieren und aufzurufen, das `MvcEvent`-Objekt zu injizieren. Die Implementierung des `EventManagerAwareInterface` sorgt übrigens dafür, dass dem Controller ein `EventManager` zur Verfügung steht, die des `ServiceLocatorAwareInterface` dafür, dass der Controller auf den `ServiceManager` zugreifen kann und damit auf die Services der Anwendung.

PostRedirectGet

Dieses Plugin unterstützt bei einer speziellen Redirect-Problemstellung im Zusammenhang mit Formularen, die per POST abgeschickt werden. Löst der Nutzer nach vorherigem Absenden eines Formulars auf dessen Bestätigungsseite einen Reload der Seite aus, werden in den meisten Situationen die POST-Daten erneut an den Server übertragen und in unglücklichen Fällen doppelte Käufe, Buchungen, usw. durchgeführt, die gar nicht gewollt waren. Um dieses Problem zu vermeiden, sollte eine Bestätigungsseite, die nach einer POST-basierten Transaktion angezeigt wird, nicht direkt im gleichen Schritt erzeugt werden. Stattdessen sendet der Server auf den erfolgreichen POST-Request zunächst einen 301/302 HTTP-Statuscode und leitet auf eine Bestätigungsseite weiter, die dann vom Browser per GET angefordert wird und damit gefahrlos auch mehrfach aufgerufen werden kann. Um diesen Mechanismus nicht selbst entwickeln zu müssen, gibt es das `PostRedirectGet`-Plugin.

Forward

Während die Redirect-Plugins über die entsprechenden HTTP-Header eine Umleitung im Client realisieren, ermöglicht das Forward-Plugin den Aufruf eines anderen Controllers aus einem Controller heraus. Im Grunde ist die Bezeichnung "Forward" gar nicht so richtig. Das Forward-Plugin

leitet nämlich nicht wirklich weiter, sondern “dispatched” lediglich einen anderen Controller. Schaut man sich die Controller des Zend Framework 2 noch einmal genau an, wird es klar: Ein Controller ist eigentlich nichts weiter, als eine Klasse, die über eine Methode `dispatch()` verfügt, ein Objekt vom Typ `Request` und eines vom Typ `Response` erwartet und damit das `DispatchableInterface` erfüllt. Was der jeweilige Controller mit dem `Request` macht, damit er später eine sinnvolle `Response` zurückgeben kann, ist ihm überlassen.

Mit dem Forward-Plugin wird demnach lediglich die `dispatch()` - Methode eines Controllers ausgeführt, der als Ergebnis - wie üblich, wenn ein Controller ausgeführt wird - ein Objekt vom Typ `ViewModel` zurückgibt. Nun bleibt es dem Controller überlassen, was er mit dem Ergebnis macht. Will man eine “Controller-Weiterleitung” emulieren, so würde dies dieser Einzeiler realisieren:

```
1 <?php
2 return $this->forward()->dispatch('Helloworld\Controller\Other');
```

Listing 9.5

Wobei “Other” in diesem Fall sinnbildlich für einen anderen Controller steht, der im Rahmen der Modul-Konfiguration dem System zuvor ebenfalls bekannt gemacht worden ist. Das `ViewModel`, das der `OtherController` erzeugt, wird hier 1-zu-1 vom ursprünglich aufrufenden Controller zurückgegeben, der erzeugt selbst gar kein `ViewModel` mehr.

Das Forward-Plugin kann aber auch dazu verwendet werden, die Ergebnisse von mehreren anderen Controllern zu aggregieren. Dazu aber mehr im Abschnitt “Konzept & Funktionsweise” des Kapitels zum Thema “View”.

Möchte man übrigens eine bestimmte Action eines anderen Controllers aufrufen, so kann man dies so realisieren:

```
1 <?php
2 return $this->forward()
3     ->dispatch(
4         'Helloworld\Controller\Other',
5         array('action' => 'test')
6     );
```

Listing 9.6

Auf diese Art und Weise lassen sich auch andere Zusatzinformationen an den aufgerufenen Controller übergeben.

URL

Mit Hilfe des URL-Plugin lässt sich auf Basis einer zuvor definierten Route und unter Angabe der Routenbezeichnung sowie ggf. weiterer Parameter eine URL erzeugen:

```
1 <?php
2 $url = $this->url()->fromRoute('routenbezeichnung', $params);
```

Listing 9.7

Params

Ermöglicht den einfachen Zugriff auf Request-Parameter, die andernfalls nur etwas umständlich zugänglich sind. Eigentlich müsste in einem Controller wie folgt etwa auf die POST-Parameter zugegriffen werden:

```
1 <?php
2 $this->getRequest()->getPost($param, $default);
```

Listing 9.8

Unter Einsatz des Plugins lässt sich der Zugriff etwas eleganter, wenn auch nicht viel kürzer gestalten:

```
1 <?php
2 $this->params()->fromPost('param', $default);
```

Listing 9.9

Lässt man die Parameterbezeichnung weg, so bekommt man alle Parameter in Form eines assoziativen Arrays zurück:

```
1 <?php
2 $this->params()->fromPost();
```

Listing 9.10

Wichtig ist, die richtige Quelle anzugeben. Die folgenden Aufrufe sind möglich:

```
1 <?php
2 $this->params()->fromPost();
3 $this->params()->fromQuery();
4 $this->params()->fromRoute();
5 $this->params()->fromFiles();
6 $this->params()->fromHeader();
```

Listing 9.11

Layout

Über das Layout-Plugin kann jederzeit das zu verwendende Layout konfiguriert werden:

```
1 <?php
2 $this->layout('layout/mein-layout');
```

Listing 9.12

Das Layout-Plugin ist etwa hilfreich, wenn man im Rahmen einer bestimmten Action ein alternatives Layout verwenden will.

Flash Messenger

Über den Flash-Messenger lassen sich Nachrichten für einen Nutzer über einen Seitenwechsel hinweg transportieren. Technisch wird dazu auf Serverseite eine Session erzeugt und die jeweilige Nachricht so kurzzeitig persistiert. Es lässt sich wie folgt eine Nachricht für einen Nutzer zum Flash-Messenger hinzufügen:

```
1 <?php
2 $this->flashMessenger()->addMessage('Der Datensatz wurde gelöscht');
```

Listing 9.13

Auf der Zielseite kann man die Nachricht (ggf. auch mehrere) wie folgt zurückholen und anzeigen:

```
1 <?php
2 $flashMessenger = $this->flashMessenger();
3
4 if ($flashMessenger->hasMessages()) {
5     $return['messages'] = $flashMessenger->getMessages();
6 }
```

Listing 9.13a

9.3 Ein eigenes Controller Plugin schreiben

Ein “Controller Plugin” ist im Zend Framework 2 eigentlich nichts besonderes, sondern eine mehr oder weniger “normale” Klasse, die eigentlich nur einen “getter” und einen “setter” erwartet, über den der Controller, in der das jeweilige “Controller Plugin” gerade Anwendung findet, injiziert wird. Die Implementierung dieser Methoden kann man sich aber sparen, wenn man das eigene “Controller Plugin” von der Klasse `Zend\Mvc\Controller\Plugin\AbstractPlugin` ableitet. Dann muss man lediglich die Methode `__invoke()` mit Leben füllen, um das “Controller Plugin” auf einfache Art und Weise in einem Controller einsetzen zu können:

```
1  <?php
2
3  namespace HelloWorld\Controller\Plugin;
4
5  use Zend\Mvc\Controller\Plugin\AbstractPlugin;
6
7  class CurrentDate extends AbstractPlugin
8  {
9      public function __invoke()
10     {
11         return date('d.m.Y');
12     }
13 }
```

Listing 9.14

Dieses Controller Plugin sorgt dafür, dass das aktuelle Datum erzeugt wird (zugegebenermaßen bräuchte man dafür eigentlich kein “Controller Plugin”). Die Klassendefinition liegt im Modul HelloWorld unter src/HelloWorld/Controller/Plugin/CurrentDate.php. Die index-Action des IndexController ist wie folgt angepasst, damit er CurrentDate nutzt und das Ergebnis an die View liefert:

```
1  <?php
2  // [...]
3  public function indexAction()
4  {
5      return new ViewModel(
6          array(
7              'greeting' => $this->greetingService->getGreeting(),
8              'date' => $this->currentDate()
9          )
10     );
11 }
```

Listing 9.15

CurrentDate kann ganz einfach aufgerufen werden, als wäre es eine Methode des aktuellen Controllers (via \$this). Damit dieser Aufruf klappt, müssen wir CurrentDate zuvor allerdings noch im ControllerPluginManager bekannt machen. Dies können wir entweder in der module.config.php des Moduls machen, oder in dessen Module-Klasse:

```
1  <?php
2  // [...]
3  public function getControllerPluginConfig()
4  {
5      return array(
6          'invokables' => array(
7              'currentDate'
8                  => 'Helloworld\Controller\Plugin\CurrentDate'
9          )
10     );
11 }
```

Listing 9.16

Einmal registriert, lässt sich dieses “Controller Plugin” in allen Controllern aufrufen. Übrigens ist `CurrentDate` nicht auf das `Helloworld`-Modul beschränkt, sondern lässt sich auch in Controllern anderer Module verwenden. Ob das tatsächlich sinnvoll ist oder nicht, ist sicherlich vom konkreten Fall abhängig.

10 Views

10.1 Konzept & Funktionsweise

Die View, im Rahmen des MVC-Patterns oft auch als “Präsentationsschicht” bezeichnet, ist für die Darstellung des Verarbeitungsergebnisses verantwortlich. Vom Moment der erfolgreichen Ausführung eines Controllers (oder Action) bis zur Darstellung des finalen Ergebnisses im Browser des Anwenders, nimmt die Ergebnis-Darstellung eine Menge unterschiedlicher Formen an und unterliegt einer ganzen Reihe von Transformationsprozessen, teils noch auf dem Server, teils dann im Client. Die initiale Repräsentation des Verarbeitungsergebnisses, das sogenannte “View Model”, erzeugt ein Controller der Anwendung. Das “View Model” hält zunächst noch keine Darstellungsinformationen (etwa HTML), sondern lediglich das “Daten-Grundgerüst” in Form von Key-Value-Paaren:

```
1  <?php
2  public function indexAction()
3  {
4      return new ViewModel(
5          array(
6              'event' => 'Beatsteaks',
7              'place' => 'Berlin',
8              'date' => $this->currentDate()
9          )
10     );
11 }
```

Listing 10.1

Alternativ kann auch schlicht ein PHP-Array zurückgegeben werden:

```
1  <?php
2  // [...]
3  public function indexAction()
4  {
5      return array(
6          'event' => 'Beatsteaks',
7          'place' => 'Berlin',
8          'date' => $this->currentDate()
9      );
10 }
```


Listing 10.2

Durch den `CreateViewModelListener` des Frameworks, der standardmäßig auf das `dispatch` - Ereignis eines `ActionControllers` reagiert, wird das erzeugte Array im Anschluss automatisch in ein `ViewModel` umgewandelt.

Das `ViewModel` ist aber mehr, als nur der Container für die Nutzdaten. Es hält außerdem die Information darüber, mit welchem Template die Daten später zu vereinen sind. Auch diese Information wird im Nachhinein durch das Framework und den `InjectViewModelListener`, ebenfalls im Rahmen des `dispatch` - Ereignis eines `ActionControllers` ausgeführt, hinzugefügt.

`ViewModel` lassen sich auch verschachteln. Diese Tatsache ist in der praktischen Anwendung sehr hilfreich und eignet sich etwa dazu, die Erzeugung einer Seite auf mehrere Controller zu verteilen. Das Controller-Plugin “Forward”, das wir bereits kennengelernt haben, kann dazu verwendet werden, um innerhalb einer Request-Verarbeitung nicht nur einen, sondern bei Bedarf gleich eine ganze Reihe von Controllern bzw. Actions auszuführen. Auf diese Weise können mehrere `ViewModel` erzeugt und in einem Atemzug verarbeitet werden:

```
1  <?php
2  // [...]
3  public function indexAction()
4  {
5      $widget = $this->forward()
6              ->dispatch('Helloworld\Controller\Widget');
7
8      $page = new ViewModel(
9          array(
10             'greeting' => $this->greetingService->getGreeting(),
11             'date' => $this->currentDate()
12          )
13      );
14
15      $page->addChild($widget, 'widgetContent');
16      return $page;
17 }
```

Listing 10.3

In dieser `indexAction` sorgen wir zunächst dafür, dass der `WidgetController` ausgeführt wird (bzw. dessen `indexAction`). Das zurückgegebene `ViewModel` wird in der Variable `$widget` zwischengespeichert. In `$page` speichern wird die Referenz auf das eigentliche `ViewModel` der `indexAction`. Bevor es allerdings zurückgegeben wird, hängt `addChild` das erzeugte “View Model” des `WidgetController` an dieses, nennen wir es der Einfachheit “primäre View Model”, an. So kann über den Schlüssel `widgetContent` auf das Ergebnis des Renderings des “View Model” des `WidgetControllers` zugegriffen werden:

```
1 // [...]
2 <sidebar>
3     <?php echo $this->widgetContent ?>
4 </sidebar>
5 // [...]
```

Listing 10.4

10.2 Layouts

Kennt man die Möglichkeit, “View Models” zu verschachteln, kann man sich die Funktionsweise von Layouts schnell erschließen. Aber vielleicht noch einmal einen Schritt zurück: Die Idee von Layouts ist es, HTML-Code (darauf reduzieren wir es an dieser Stelle der Einfachheit halber einmal; natürlich können aber auch Datenstrukturen jenseits von HTML mit dem Zend Framework 2 erzeugt werden) aufzunehmen, der von vielen oder gar allen Controllern und Actions benötigt wird. Dazu gehören META-Tags, das HTML-Grundgerüst, Referenzen zu CSS-Dateien und ähnliches.

Ein Layout ist technisch gesehen nichts anderes, als ein `ViewModel`, dass das durch eine Controller-Action erzeugte `ViewModel` als “Child” referenziert, ebenso wie in dem Beispiel oben das `ViewModel` einer Controller-Action jenes einer anderen Controller-Action referenziert hat. Für den Zugriff auf das `ViewModel` der Controller-Action innerhalb des Layout-Templates registriert das Framework automatisch den Schlüssel `content`, so wie wir im Beispiel zuvor händisch den Schlüssel `widgetContent` erzeugt haben. Im Layout-Template kann also auf folgende Weise auf das Ergebnis einer Controller-Action zugegriffen werden:

```
1 <html>
2     <head>
3         <title>Meine Seite</title>
4     </head>
5     <body>
6         <?php echo $this->content; ?>
7     </body>
8 </html>
```

Listing 10.5

Das vom Framework automatisch herangezogene Layout-Template lässt sich über das im vorherigen Kapitel besprochene “Controller Plugin” steuern oder aber über einen “View Helper”, den wir uns im Folgenden ebenfalls noch genauer ansehen werden.

10.3 View Helper

Häufig gibt es den Bedarf, die Daten des `ViewModel` im Zuge des Rendering noch weiter aufzubereiten oder zusätzliche Daten zu erzeugen. So gibt es etwa rund um die Themen “Navigation”, “META-Tags” und ähnlichem eine ganze Reihe wiederkehrender, View-bezogener Aufgaben, die man einmalig bewältigen und in Form eines “View Helpers” wiederverwendbar machen kann. Das Zend Framework 2 kommt standardmäßig mit einer ganzen Reihe von fertigen “View Helpers”, die man sofort einsetzen kann. Zusätzlich kann man seine eigenen View Helper schreiben.

10.4 Einen eigenen View Helper schreiben

Ein “View Helper” ist im Zend Framework 2 eigentlich nichts besonderes, sondern eine mehr oder weniger “normale” Klasse, die eigentlich nur einen “getter” und einen “setter” erwartet, über den die View, in der der jeweilige “View Helper” gerade Anwendung findet, injiziert wird. Die Implementierung dieser Methoden kann man sich aber sparen, wenn man den eigenen “View Helper” von der Klasse `Zend\View\Helper\AbstractHelper` ableitet. Dann muss man lediglich die Methode `__invoke()` mit Leben füllen, um den “View Helper” auf einfache Art einsetzen zu können:

```
1  <?php
2
3  namespace HelloWorld\View\Helper;
4
5  use Zend\View\Helper\AbstractHelper;
6
7  class DisplayCurrentDate extends AbstractHelper
8  {
9      public function __invoke()
10     {
11         return date('d.m.Y');
12     }
13 }
```

Listing 10.6

Dieser “View Helper” sorgt dafür, dass in einer View das aktuelle Datum ausgegeben werden kann (zugegebenermaßen bräuchte man dafür eigentlich keinen “View Helper”). Die Klassendefinition liegt im Modul `HelloWorld` unter `src/HelloWorld/View/Helper/DisplayCurrentDate.php`. Die Datei `index.phtml`, also die View zur Action `index` im Controller `Index`, ist wie folgt angepasst und nutzt nun diesen “View Helper”:

```
1 <h1><?php echo $this->greeting; ?></h1>
2 <h2><?php echo $this->displayCurrentDate(); ?></h2>
```

Listing 10.7

DisplayCurrentDate kann ganz einfach aufgerufen werden, als wäre er eine Methode der aktuellen View (via \$this). Damit dieser Aufruf klappt, müssen wir DisplayCurrentDate zuvor allerdings noch im ViewHelperManager bekannt machen. Dies können wir entweder in der module.config.php des Moduls machen, oder in dessen Module-Klasse:

```
1 <?php
2 public function getViewHelperConfig()
3 {
4     return array(
5         'invokables' => array(
6             'displayCurrentDate'
7                 => 'Helloworld\View\Helper\DisplayCurrentDate'
8         )
9     );
10 }
```

Listing 10.8

oder

```
1 <?php
2 'view_helpers' => array(
3     'invokables' => array(
4         'displayCurrentDate'
5             => 'Helloworld\View\Helper\DisplayCurrentDate'
6     )
7 )
8 // [...]
```

Listing 10.9

Einmal registriert, lässt sich dieser “View Helper” in allen Views aufrufen. Übrigens ist DisplayCurrentDate nicht auf das Helloworld-Modul beschränkt, sondern lässt sich auch in Views anderer Module verwenden. Ob das tatsächlich sinnvoll ist oder nicht, ist sicherlich vom konkreten Fall abhängig.

11 Model

Was genau ist eigentlich das “Model” einer Anwendung? Während “View” und “Controller” inhaltlich relativ klar definiert und vom Framework in Form und Funktion ausgestaltet sind, ist dies beim “Model” nicht so offensichtlich. Das liegt vor allem an der Tatsache, dass ein “Model” je nach Anwendungstyp und Situation eine sehr unterschiedliche Gestalt annehmen kann. So haben wir ja bereits das “View Model” kennengelernt. Ist dies das “Model”, dass sich in “MVC” verbirgt? Nein, allerdings hat es einige Ähnlichkeiten mit dem “Model”, das eigentlich gemeint ist.

Gehen wir zunächst einmal definitorisch an die Sache ran. [Wikipedia](http://de.wikipedia.org/wiki/Modell)¹ definiert ein Modell wie folgt:

1. Abbildung – Ein Modell ist stets ein Modell von etwas, nämlich Abbildung, Repräsentation eines natürlichen oder eines künstlichen Originals, das selbst wieder Modell sein kann.
2. Verkürzung – Ein Modell erfasst im Allgemeinen nicht alle Attribute des Originals, sondern nur diejenigen, die dem Modellschaffer bzw. Modellnutzer relevant erscheinen.
3. Pragmatismus – Modelle sind ihren Originalen nicht per se eindeutig zugeordnet. Sie erfüllen ihre Ersetzungsfunktion a) für bestimmte Subjekte (Für Wen?), b) innerhalb bestimmter Zeitintervalle (Wann?) und c) unter Einschränkung auf bestimmte gedankliche oder tätliche Operationen (Wozu?).

Ein “Model” ist grundsätzlich also eine vereinfachte Repräsentation der Realität, die zum entsprechenden Zeitpunkt ausreichend für den jeweiligen Anwendungszweck ist. So ist das “View Model” die vereinfachte Repräsentation einer Webseite, beschränkt auf die anzuzeigenden Daten, Informationen zum Status bestimmter Aktionselemente (Button aktiv oder inaktiv, Box aufgeklappt oder zugeklappt, usw.) sowie der Referenz zu einem Template, mit dem diese Daten später verknüpft werden sollen. Im Rahmen des Renderings entsteht aus dem “View Model” ein weiteres, wenn man so will “höher entwickeltes” und sich näher an der Realität befindliches Abbild der Realität: Das fertig erzeugte Markup zur späteren Darstellung einer Website. Diese wird an den Aufrufer zurückgeliefert und der Browser erzeugt daraus ein weiteres Modell, das [DOM](http://de.wikipedia.org/wiki/Document_Object_Model)². Auf Basis des “Document Object Model (DOM)”, das passenderweise den Term “Model” sogar bereits in sich trägt, werden dann in einem abschließenden Schritt bunte Pixel auf den Bildschirm gezaubert. Und auch dieses Ergebnis ist eigentlich wieder nur ein Modell. Und auch diese Erklärung selbst ist nur ein Modell, denn die in meinen Augen an dieser Stelle irrelevanten Informationen und Details habe ich ausgelassen.

Bevor es aber allzu philosophisch wird nun aber zum “Model” von “MVC”. In der engeren Definition ist dieses “Model” das Abbild, der von der Anwendung berührten Realität, also die fachliche

¹<http://de.wikipedia.org/wiki/Modell>

²http://de.wikipedia.org/wiki/Document_Object_Model

Domäne, um die es sich jeweils dreht. Etwa die Abbildung von E-Commerce-Prozessen in Online-Shops oder -Marktplätzen, die Verwaltung von Kunden, Ansprechpartnern und Vorfällen in CRM-Systemen oder Dokumenten, Autoren und Rubriken in CMS-Systemen. Um nur einmal einige wenige Beispiele zu nennen. Dieses fachliche Modell ist also im Kern vollkommen anwendungsspezifisch und lässt sich schlußendlich generisch auch gar nicht genauer definieren. Demnach ist auch die Gestalt des fachlichen Modells sowie dessen technische Ausprägung nicht auf eine bestimmte Art und Weise festgelegt. Sucht man nach Standards im Bereich der “Enterprise Applications” oder typischer Webanwendungen, für die das Zend Framework 2 ja primär gemacht ist, bietet sich die Definition des [Domain-Driven Design](http://de.wikipedia.org/wiki/Domain-Driven_Design)³ an. Dort zählen die folgenden Einheiten einer Anwendung zum fachlichen Modell:

- Entites
- Repositories
- Value Objects
- Aggregates
- Assoziationen
- Business-Services
- Business-Events
- Factories

Dies soll keine Einführung in die Welt des “Domain-Driven Design” werden, insbesondere nicht, weil es eben nur einen Weg darstellt, um das “Model” der eigenen Anwendung zu realisieren und daher keine Allgemeingültig besitzt. Dennoch korrelieren einige Konstrukte des “Domain-Driven Design” mit denen des Zend Frameworks, bei denen sich ein genauerer Blick lohnt.

11.1 Entities, Repositories & Value Objects

Eine “Entity” repräsentiert ein eindeutiges Objekt in einem System, etwa ein Kunde oder eine Bestellung in einem Shop-System. Erreicht wird die Eindeutigkeit in aller Regel durch die Nutzung von IDs, etwa einer eindeutigen Kundennummer. Manchmal lassen sich auch die Eigenschaften eines Objekts so kombinieren, dass die Eindeutigkeit gegeben ist. Allerdings ist die z.B. bei natürlichen Personen häufig sehr schwierig anderweitig zu erreichen. Denn etwa in einer Stadt wie Berlin, München oder Hamburg gibt es mehrere Menschen, die den exakt gleichen Namen tragen. Manche von ihnen haben zudem sogar am gleichen Tag Geburtstag. Dennoch handelt es konzeptionell um unterschiedliche “Entitäten”, die man auf jeden Fall auseinander halten muss.

Die Bezeichnung “Entity” taucht insbesondere auch wieder im Zusammenhang mit der sog. “Persistenz” auf, also der Speicherung jener Objekte in Datenbanken, so dass sie einen Request

³http://de.wikipedia.org/wiki/Domain-Driven_Design

überdauern. So verwendet das ORM-System [Doctrine 2](#)⁴ die Bezeichnung “Entity” etwa für alle Objekte, die in der Datenbank gespeichert und verwaltet werden.

Ein “Repository” ermöglicht es, die in einer Datenbank gespeicherten “Entities” wieder hervorzuholen. Je nach Implementierung fungiert ein “Repository” als Container für die für einen bestimmten Entitätstyp benötigten SQL-Abfragen. Manchmal ist es aber auch mehr, wie wir auch noch sehen werden.

Ein “Value Object” ist im Gegensatz zu einer “Entity” ohne eigene Identität. Zwei Instanzen der Klasse `\DateTime` etwa, dem PHP-Standardobjekt zur Repräsentation von Datum und Uhrzeit, die mit dem identischen Timestamp erzeugt wurden und somit die gleichen “Eigenschaften” haben, sind schlicht identisch. Eine Unterscheidung ist konzeptionell nicht erforderlich. In aller Regel besteht für ein “Value Object” auch kein Bedarf für Persistenz, wobei das technisch gesehen grundsätzlich auch denkbar wäre.

Eine “Entity” oder auch ein “Value Object” kann in PHP über eine simple Klasse abgebildet werden:

```
1  <?php
2  class User
3  {
4      private $name;
5      private $email;
6      private $password;
7
8      public function setEmail($email)
9      {
10         $this->email = $email;
11     }
12
13     public function getEmail()
14     {
15         return $this->email;
16     }
17
18     public function setName($name)
19     {
20         $this->name = $name;
21     }
22
23     public function getName()
24     {
25         return $this->name;
26     }
```

⁴doctrine-project.org

```
27
28     public function setPassword($password)
29     {
30         $this->password = $password;
31     }
32
33     public function getPassword()
34     {
35         return $this->password;
36     }
37 }
```

Listing 11.1

Wie wir später noch im Rahmen von Zend\Db und Zend\Form sehen werden, können sich Entitäten, aber auch “Values Objects”, die auf diese Art und Weise repräsentiert werden, an verschiedenen Stellen in der Anwendung als äußerst nützlich erweisen.

11.2 Business-Services & Factories

Über Services und Factories haben wir bereits gesprochen. Ein Service kann über eine Factory - oder über Zend\Di, wie wir im nächsten Kapitel noch sehen werden - erzeugt werden. Als einen “Business-Service” bezeichnet man Funktionen, die sich auf die fachliche Domäne beziehen, sich aber nicht eindeutig einer “Entity” zuordnen lassen oder zuordnen lassen sollen. In einem Shopsystem könnte man etwa einen Service “CurrencyConversion” definieren, der ein “Value Object” mit den Eigenschaften “Wert” und “Währung” samt der Information darüber, in welche Währung eine Umrechnung stattfinden soll, übergeben bekommt. Das Umrechnungsergebnis wird in Form eines neuen “Values Objects” zurückgegeben. Grundsätzlich wäre es auch möglich, die Umrechnung als Teil des “Value Objects” zu verstehen und die Funktion in der entsprechenden Klasse, nennen wir sie hier einmal Price (der sich aus “Wert” und “Währung” definiert) zu implementieren. Allerdings müsste das entsprechende “Value Object” dann Kenntnis über die vom System unterstützten Währungen haben, die vermutlich als “Entities” in einer Datenbank abgelegt sind. Ob man diese Verbindung wirklich will, ist fraglich.

Die Funktion betrifft also nicht nur eines Typus von Objekt im System, sondern gleich mehrere. Der “CurrencyConversionService” wäre daher ein guter Kandidat, um als “Business Service” abgebildet zu werden und würde damit dann ebenfalls zu “Model” der Anwendung zählen.

Davon abgrenzen lassen sich sog. “Technische Services” oder “Infrastructure Services”, die eine nicht-fachliche Leistung erbringen, etwa den physischen Versand von E-Mails oder SMS-Nachrichten, Logging-Funktionen und so weiter. Sie drehen sich in aller Regel weniger um die fachlichen Objekte der Anwendung und stellen selbst auch keine fachliche Funktion dar. Das Zend Framework 2 bringt etwa auch bereits eine ganze Reihe von “Technischen Services” mit, die

unabhängig von der jeweiligen Fachlichkeit eingesetzt werden können. “Technische Services” zählen daher im Gegensatz zu den “Business Services” nicht zum “Model” einer Anwendung.

11.3 Business-Events

In den vorherigen Kapitel haben wir bereits das Eventsystem und die vielen Events des Frameworks im Rahmen der Requestverarbeitung kennengelernt, etwa `route`, `dispatch` und `render`. Auch haben wir mit `getGreeting` ein eigenes Event konzipiert, das nicht technisch-funktionaler Natur war, sondern primär dabei hilft, die Business-Prozesse innerhalb der Anwendung flexibel zu halten. So können wir Workflows anpassen, Aktionen hinzufügen oder entfernen und deren Ausführungsreihenfolge jederzeit ändern. Auch diese fachlichen Events sind ebenfalls Teil des “Models” der Anwendung. In einem Shopsystem wären typische Events etwa das Hineinlegen eines Artikels in den Warenkorb, der abgeschlossene Checkout oder die Registrierung einer eingegangenen Retoure über die entsprechende Administrationskonsole durch die Shop-Mitarbeiter.

12 Routing

12.1 Einleitung

Hinter der Mechanik des “Routing” steht die grundsätzliche Idee, dass sich eine URL nicht mehr 1-zu-1 auf eine existierende (PHP-)Datei abbilden lassen muss. Stattdessen können URLs frei gewählt und die entsprechende Verarbeitungslogik - in aller Regel ein Controller und eine Action - mit ihnen verknüpft werden. Nicht erst seitdem es die Disziplin der [Suchmaschinenoptimierung](http://de.wikipedia.org/wiki/Suchmaschinenoptimierung)¹ (SEO) gibt und der Online-Marketer nun dafür sorgen kann, dass die URLs der Anwendung auch für Google “sprechend” gestaltet sind und die richtigen Keywords enthalten, freut man sich über die gewonnene Flexibilität in der Ausgestaltung von URLs. Gerade auch bei lokalisierten Anwendungen, bei denen URLs in unterschiedlichen Sprachen erzeugt werden sollen, stößt man ohne ein ausgefeiltes “Routing” schnell an seine Grenzen. Das Zend Framework 2 bringt wie auch bereits sein Vorgänger eine sehr leistungsfähige und flexible “Routing”-Lösung mit - von Grund auf neu programmiert, performanter als zuvor und auch in sich schlüssiger konzipiert.

Übrigens ist das Matching einer URL auf einen Controller bereits ein Spezialfall, weil es sich gar nicht zwangsläufig um eine URL handeln muss, die als Ausgangswert für das Matching herangezogen wird. Wie wir später im Rahmen von `Zend\Console` noch sehen werden, können auch frei definierbare “Kommandos” zu Controllern zugeordnet werden. Das ist etwa für Cron-Jobs sehr praktisch. Doch dazu später mehr. Der Einfachheit halber werde ich bei den weiteren Ausführungen zum Thema “Routing” bei der URL als Ausgangswert bleiben, auch wenn die Aussagen allgemeine Gültigkeit haben und grundsätzlich nicht auf ebenjene beschränkt sind.

12.2 Definition von Routen

Als “Route” wird ein konkretes Mapping einer URL auf einen Controller bezeichnet: Wird URL “X” angefordert, führe Controller “Y” und dessen Action “Z” aus. Eine einfache Routendefinition, im Rahmen der `module.config.php`, sieht wie folgt aus:

¹<http://de.wikipedia.org/wiki/Suchmaschinenoptimierung>

```
1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'sayhello' => array(
6              'type' => 'Zend\Mvc\Router\Http\Literal',
7              'options' => array(
8                  'route' => '/sayhello',
9                  'defaults' => array(
10                     'controller' => 'helloworld-index-controller',
11                     'action' => 'index',
12                 )
13             )
14         )
15     )
16 )
17 // [...]
```

Listing 12.1

Wenn der Pfad der aufgerufenen URL /sayhello lautet, wird der helloworld-index-controller und dessen index-Action ausgeführt. Nicht viel mehr, aber auch nicht weniger. Eine einigermaßen komplexe Anwendung wird eine ganze Reihe solcher Definitionen mitbringen. Dazu gleich mehr.

Der Schlüssel helloworld-index-controller wurde in diesem Fall zuvor im Rahmen der di-Konfiguration als Alias für den eigentlichen Controller definiert (mehr zu Zend\Di weiter hinten in diesem Buch):

```
1  <?php
2  // [...]
3  'alias' => array(
4      'helloworld-index-controller'
5          => 'Helloworld\Controller\IndexController'
6  )
```

Listing 12.2

Im Framework passiert Folgendes: Der Router nimmt den Request entgegen und geht durch die Liste aller hinterlegten “Routen”. Dies passiert entweder in Form eines “Stacks”, sprich, die zuletzt hinzugefügte “Route” wird zuerst auf Übereinstimmung überprüft, die Erste zuletzt, oder aber die “Routen” werden in Form eines Baumes abgelegt. In jedem Fall führt ein Treffer dazu, dass ein Objekt vom Typ RouteMatch erzeugt, zurückgegeben und die Arbeit des Routers beendet wird. Die vom Router in das RouteMatch-Objekt übertragenen Werte, unter anderem der für diese “Route” konfigurierte Controller, wird in der weiteren Verarbeitung des Requests dann ausgewertet. So weiß etwa der Dispatcher bescheid, welchen Controller es zu instanzieren gilt.

12.3 Test auf Übereinstimmung

Das Framework bringt eine Reihe von Möglichkeiten mit, um einfache wie komplexe, etwa verschachtelte Übereinstimmungsregeln zu definieren.

Pfad überprüfen

Die einfachste Variante, die sog. Literal-Route, haben wir oben bereits gesehen: Es wird eine Zeichenkette definiert, die 1-zu-1 dem Pfad der angeforderten URL entsprechend muss. Ist dies der Fall, so “greift” diese Route und basierend auf dessen Konfiguration werden die nötigen Hebel in Bewegung gesetzt.

Eine flexiblere Option ist die Regex-Route, bei der der Pfad der URL auf einen regulären Ausdruck passen muss. Nehmen wir hierfür ein konkretes Beispiel: Ein Online-Shop für Schuhe könnte etwa den folgenden URL-Pfad für Detailseiten verwenden, wobei der vordere Teil den “Slug” der Produktbezeichnung darstellt, während die Zahl danach die Artikelnummer repräsentiert: `/converse-as-ox-can-sneaker-black/373682726`. Eine Regex-Route, die für diese Art von URL funktionieren würde, sieht wie folgt aus:

```
1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'detailPage' => array(
6              'type' => 'Zend\Mvc\Router\Http\Regex',
7              'options' => array(
8                  'regex' => '/(<slug>[a-zA-Z0-9_-]+)/(<id>[0-9]+)',
9                  'spec'  => '/%slug%/%id%',
10                 'defaults' => array(
11                     'controller' => 'helloworld-index-controller',
12                     'action'     => 'index',
13                 )
14             )
15         )
16     )
17 )
18 // [...]
```

Listing 12.3

Im Controller kann dann wie folgt etwa auf den Wert für “Slug” zugegriffen werden:

```

1 <?php
2 $this->getEvent()->getRouteMatch()->getParam('slug');

```

Listing 12.4

Werfen wir noch einmal einen genauen Blick auf den regulären Ausdruck `/(<slug>[a-zA-Z0-9_-]+)/(?<id>[0-9]+)`: Der Pfad muss zunächst mit einem “/” beginnen. Danach können beliebig viele Ziffern, Buchstaben, der Unterstrich und der Bindestrich folgen, mindestens aber muss es eines dieser Zeichen an dieser Stelle geben (+). Alles, was dort zwischen dem “/” davor und dem “/” danach folgt, wird als “slug” bezeichnet und entsprechend auch so später über das `RouteMatch`-Objekt bereitgestellt. Nach dem verpflichtenden zweiten “/” können eine oder mehrere Ziffern zwischen 0 und 9 folgen. Diese Zahl wird als “id” bezeichnet und ebenfalls im `RouteMatch`-Objekt bereitgestellt.

Die Option “spec” schauen wir uns später noch einmal genauer an. Sie dient für den “Rückweg”, also der Erzeugung einer URL auf Basis ebenjener Route.

Alternativ dazu hätte man die Herausforderung auch über eine Segment-Route lösen können:

```

1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'detailPage' => array(
6             'type' => 'Zend\Mvc\Router\Http\Segment',
7             'options' => array(
8                 'route' =>('/:slug/:id',
9                 'defaults' => array(
10                    'controller' => 'helloworld-index-controller',
11                    'action' => 'index',
12                )
13            )
14        )
15    )
16 )
17 // [...]

```

Listing 12.5

Auf die Option “spec” können wir hier verzichten, weil sie sich durch die “route” ergibt. Es können zudem auch optionale Segmente definiert werden, wenn man diese in eckige Klammern setzt, z.B. “[/:id]”. Im Controller kann dann ebenfalls wie folgt etwa auf den Wert für “Slug” zugegriffen werden:

```
1 <?php
2 $this->getEvent()->getRouteMatch()->getParam('slug');
```

Listing 12.6

Durch die Hinzunahme von “Constraints” lassen sich auch bei der Segment-Route auf regulären Ausdrücken basierende Überprüfungen, in diesem Fall aber der einzelnen Segmente, realisieren:

```
1 <?php
2 // [...]
3 'constraints' => array(
4     'slug' => '[a-zA-Z0-9_-]+',
5     'id'    => '[0-9]+'
6 )
7 // [...]
```

Listing 12.7

Hostname, Protokoll & Co. überprüfen

Auch auf einen Hostnamen kann mit Hilfe einer Hostname-Route geprüft werden:

```
1 <?php
2 // [...]
3 'router' => array(
4     'routes' => array(
5         'detailPage' => array(
6             'type' => 'Zend\Mvc\Router\Http\Hostname',
7             'options' => array(
8                 'route' => 'blog.meinedomain.de',
9                 'defaults' => array(
10                    'controller' => 'helloworld-index-controller',
11                    'action' => 'index',
12                )
13            )
14        )
15    )
16 )
17 // [...]
```

Listing 12.8

Hat man nur diese Regel erstellt, spielt die Pfadangabe in der URL gar keine Rolle. Jede URL, die den Hostnamen `blog.meinedomain.de` enthält, wird auf den angegebenen Controller abgebildet.

Über die Scheme-Route lässt sich Überprüfen, ob “https” verwendet wird - sehr praktisch, wenn man gewisse Inhalte, etwa das “Kundenkonto”, nur über https zugänglich machen will.

Über die Method-Route lässt sich der “Modus”, in dem der HTTP-Request abgesendet wurde, also etwa “POST”, “GET” oder “PUT”, als Basis für die Überprüfung verwenden. Mit Hilfe dieser Route lassen sich z.B. [REST²](http://de.wikipedia.org/wiki/Representational_State_Transfer)-Webservices strukturieren (dafür gibt es aber, wie wir später noch sehen werden, mit dem `AbstractRestController` eine noch bessere Lösung) oder elegant die Formularanzeige und -verarbeitung separieren:

```
1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'blog' => array(
6              'type' => 'Zend\Mvc\Router\Http\Literal',
7              'options' => array(
8                  'route' => '/contactform',
9              ),
10             'child_routes' => array(
11                 'formShow' => array(
12                     'type' => 'method',
13                     'options' => array(
14                         'verb' => 'get',
15                         'defaults' => array(
16                             'controller' => 'form-controller',
17                             'action' => 'show',
18                         )
19                     )
20                 ),
21                 'formProcess' => array(
22                     'type' => 'method',
23                     'options' => array(
24                         'verb' => 'post',
25                         'defaults' => array(
26                             'controller' => 'form-controller',
27                             'action' => 'process',
28                         )
29                     )
30                 )
31             )
32         )
33     )
34 )
```

²http://de.wikipedia.org/wiki/Representational_State_Transfer

```

31         )
32     )
33 )
34 )
35 // [...]

```

Listing 12.9

Hier wird immer dann, wenn es sich um einen GET-Request auf die URL `/contactform` handelt, die `show`-Action des `form-controller` ausgeführt und wenn es sich um einen POST-Request handelt die `process`-Action. Andernfalls müsste man eine unschöne `isPost()`-Logik in der Action selbst implementieren, um zwischen initialer Formulardarstellung (GET) und Formularverarbeitung (POST) unterscheiden zu können (im Praxisteil werden wir für die Verarbeitung von Formularen auch noch eine schöne Lösung auf Basis eines speziellen Controller-Typen entwickeln).

Regeln kombinieren

Manche Regeln machen für sich allein genommen kaum Sinn. Auf den Hostnamen zu überprüfen, ohne sich in irgendeiner Form um die Pfadangabe zu kümmern, ist meistens nicht hilfreich. Daher lassen sich Regeln in Form von Bäumen miteinander kombinieren, wie man bereits beim letzten Beispiel sehen konnte. Nehmen wir noch ein weiteres Beispiel: Unter `www.meinedomain.de` betreiben wir einen Online-Shop und unter `blog.meinedomain.de` den passenden Blog dazu. Unter der URL `blog.meinedomain.de/testberichte` sollen die aktuellsten Testberichte angezeigt werden, die URL `www.meinedomain.de/testberichte` gibt es hingegen nicht, muss also einen 404-Fehler erzeugen. Dazu lassen sich eine `Hostname`-Route und eine `Literal`-Route kombinieren:

```

1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'blog' => array(
6              'type' => 'Zend\Mvc\Router\Http\Hostname',
7              'options' => array(
8                  'route' => 'blog.meinedomain.de',
9                  'defaults' => array(
10                     'controller' => 'helloworld-index-controller',
11                     'action' => 'index',
12                 )
13             ),
14             'child_routes' => array(
15                 'tests' => array(
16                     'type' => 'literal',

```



```

17         'options' => array(
18             'route'      => '/testberichte',
19             'defaults' => array(
20                 'controller' =>
21                     'helloworld-index-controller',
22                 'action'      => 'tests',
23             )
24         )
25     )
26 )
27 )
28 )
29 )
30 // [...]

```

Listing 12.10

Nur dann, wenn der URL-Pfad `/testberichte` lautet und gleichzeitig der Hostname auf `blog.meinedomain.de`, greift diese Route. Dies ist durch den Abschnitt `child_routes` definiert. Eine `child_route` kann seinerseits wieder eigene `child_routes` haben, so dass eine ganze Baumstruktur entsteht. Gibt es mehrere `child_routes` auf der gleichen "Ebene", sind sie als Alternativen zu verstehen:

```

1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'blog' => array(
6              'type' => 'Zend\Mvc\Router\Http\Hostname',
7              'options' => array(
8                  'route' => 'blog.meinedomain.de',
9                  'defaults' => array(
10                     'controller' => 'helloworld-index-controller',
11                     'action' => 'index',
12                 )
13             ),
14             'child_routes' => array(
15                 'tests' => array(
16                     'type' => 'literal',
17                     'options' => array(
18                         'route' => '/testberichte',
19                         'defaults' => array(
20                             'controller' => 'tests-controller',
21                             'action' => 'show',

```

```
22         )
23     )
24 ),
25     'testArchive' => array(
26         'type'      => 'literal',
27         'options' => array(
28             'route'    => '/testberichte/archiv',
29             'defaults' => array(
30                 'controller' => 'tests-controller',
31                 'action'     => 'archive',
32             )
33         )
34     )
35 )
36 )
37 )
38 )
39 // [...]
```

Listing 12.11

Hier sind jetzt also die Routen für die URL-Pfade `/testberichte` und `/testberichte/archiv` unabhängig voneinander definiert, allerdings setzen beide voraus, dass der Hostname auf `blog.meinedomain.de` lautet.

12.4 Erzeugung von URLs

Will man einen Link in einer Anwendung erzeugen, der auf eine interne Seite verweist, die auch über eine definierte “Route” zu erreichen ist, kann man sich die für das “href”-Attribut notwendig URL auf einfache Art und Weise erzeugen lassen. Der Vorteil gegenüber der händischen Erzeugung der URL ist, dass man später bei Bedarf an zentraler Stelle die Form einer URL anpassen kann, ohne durch die gesamte Anwendung pflügen und alle Links individuell anpassen zu müssen.

Schauen wir uns noch einmal die (Regex-)Routendefinition an:

```

1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'detailPage' => array(
6              'type' => 'Zend\Mvc\Router\Http\Regex',
7              'options' => array(
8                  'regex' => '/(<slug>[a-zA-Z0-9_-]+)/(<id>[0-9]+)',
9                  'spec' => '/%slug%/<id>',
10                 'defaults' => array(
11                     'controller' => 'helloworld-index-controller',
12                     'action' => 'index',
13                 )
14             )
15         )
16     )
17 )
18 // [...]

```

Listing 12.12

Der für die Erzeugung einer URL auf Basis dieser Routendefinition interessante Teil ist die “spec”. Dort wird definiert, wie sich eine URL zu dieser Seite schematisch zusammensetzt. Einmal definiert, kann z.B. über das “Controller-Plugin” `url` eine URL generiert werden:

```

1  <?php
2  // [...]
3  $href = $this->url()
4          ->fromRoute(
5              'detailPage',
6              array("slug" => "adidas-samba-sneaker", "id" => 34578347)
7          );

```

Listing 12.13

Wichtig ist hierbei, den richtigen Namen für die “Route” zu verwenden und alle notwendigen URL-Bestandteile zu übergeben.

12.5 Standard-Routing

Schon aus der Version 1 des Zend Framework kennt und schätzt man das “Standard-Routing”, das ein wenig Komfort zu Lasten der Flexibilität bringt. Das “Standard-Routing” sorgt dafür, dass der Pfad

in der URL standardmäßig auf ein Modul, einen Controller und eine Action abgebildet wird. Die URL `/blog/entry/add` würde also die Add-Action, im Entry-Controller des Blog-Moduls anstoßen.

War dieses “Standard-Routing” seinerzeit noch in die MVC-Mechaniken der Version 1 des Frameworks “eingebacken”, ist dies mit Version 2 nicht mehr der Fall. Ein “Standard-Routing” gibt es in dieser Form also eigentlich gar nicht mehr, lässt sich aber mühelos emulieren, wenn man die vorherigen Abschnitte aufmerksam gelesen hat. Über die folgende Definition ermöglichen wir das “Standard-Routing” für das HelloWorld-Modul:

```

1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'helloworld' => array(
6              'type'      => 'Literal',
7              'options' => array(
8                  'route'      => '/helloworld',
9                  'defaults' => array(
10                     '__NAMESPACE__' => 'Helloworld\Controller',
11                     'controller'    => 'Index',
12                     'action'       => 'index',
13                 ),
14             ),
15             'may_terminate' => true,
16             'child_routes' => array(
17                 'default' => array(
18                     'type'      => 'Segment',
19                     'options' => array(
20                         'route'      => '[:controller[/:action]]',
21                         'constraints' => array(
22                             'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
23                             'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
24                         ),
25                         'defaults' => array(
26                             )
27                     )
28                 )
29             )
30         )
31     )
32 )
33 // [...]
```

Listing 12.14

Allerdings müssen dennoch weiterhin die entsprechenden Controller zunächst im System bekannt gemacht werden (das war in der Version 1 des Frameworks auch nicht der Fall):

```
1  <?php
2  // [...]
3  'controllers' => array(
4      'invokables' => array(
5          'Helloworld\Controller\Widget'
6              => 'Helloworld\Controller\WidgetController',
7          'Helloworld\Controller\Index'
8              => 'Helloworld\Controller\IndexController'
9      )
10 )
```

Listing 12.15

So lassen sich nun etwa die URLs `/helloworld` und `/helloworld/widget/index` aufrufen.

Noch zwei kurze Hinweise zu der Routendefinition: Die `may_terminate`-Konfiguration weist den Router, der die Regeln ja auswertet, darauf hin, dass auch bereits `/helloworld` alleine, für sich genommen, eine valide Route darstellt und nicht zwangsläufig noch die Definitionen der `child_routes` hinzugezogen werden müssen. Lässt man `may_terminate` weg (und setzt es damit implizit auf `false`), würde `/helloworld` eben nicht mehr funktionieren, sondern nur noch `/helloworld/widget/index` (wenn wir hier bei unserem Beispiel von oben bleiben). Und mithilfe der Option `'__NAMESPACE__'` wird jeweils für den aus der URL “herausgeschnittenen” Controller der vollqualifizierte Klassenname gebildet, so dass er auf die hinterlegte Controller-Konfiguration passt. Andernfalls wird etwa bei der URL `/helloworld/widget/index` vergebens nach einem Controller gesucht, der unter dem Namen “Widget” bekannt ist. Sein korrekter Name lautet ja aber eben `Helloworld\Controller\Widget`, der auf diesem Wege so automatisch richtig erzeugt wird.

12.6 Kreatives Routing: A/B-Tests

Das Routing lässt sich mit ein wenig Kreativität auch für Dinge nutzen, an die man zunächst gar nicht denken würde. So kann man etwa einen Teil des Traffics einer URL, in diesem Fall (theoretisch) die Hälfte, auf eine alternative Implementierung routen:

```
1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'detailPage' => array(
6              'type' => 'Zend\Mvc\Router\Http\Literal',
7              'options' => array(
8                  'route' => '/beispielseite',
9                  'defaults' => array(
10                     'controller' => 'helloworld-index-controller',
11                     'action' => rand(0,1) ? 'original' : 'variation'
12                 )
13             ),
14         )
15     )
16 )
17 // [...]
```

Listing 12.16

Hier muss es jetzt im helloworld-index-controller eine Action original und eine Action variation geben, die jeweils einmal die Ausgangsversion einer Seite und einmal eine hypothetisch verbesserte Variante dieser Seite enthält, die sich im Rahmen eines [A/B-Tests](http://de.wikipedia.org/wiki/A/B-Test)³ als etwa in der [Konversionsrate](http://de.wikipedia.org/wiki/Konversion_(Marketing))⁴ überlegen behaupten muss. In Zusammenarbeit mit einem [Webtracking](http://de.wikipedia.org/wiki/Web_Analytics)⁵-Tool, wie etwa dem kostenlosen [Google Analytics](http://www.google.com/intl/de/analytics/)⁶, lassen sich so auch komplexe A/B-Test-Szenarien aufbauen und auswerten.

³<http://de.wikipedia.org/wiki/A/B-Test>

⁴[http://de.wikipedia.org/wiki/Konversion_\(Marketing\)](http://de.wikipedia.org/wiki/Konversion_(Marketing))

⁵http://de.wikipedia.org/wiki/Web_Analytics

⁶<http://www.google.com/intl/de/analytics/>

13 Dependency Injection

13.1 Einführung

Die Idee des “Dependency Injection (DI)” ist die Folgende: Eine Objekt holt sich nicht selbst weitere Objekte, die es benötigt, sondern bekommt sie von Außen übergeben, sozusagen “injiziert”. Der größte Vorteile an diesem Vorgehen ist die Tatsache, dass das abhängige Objekt selbst nicht mehr wissen muss, wovon es eigentlich genau abhängig ist. Das Wissen über diese Abhängigkeit wird aus dem abhängigen Objekt extrahiert und separat verwaltet. Dies erlaubt es dem Anwendungsentwickler, dem abhängigen Objekt in bestimmten Situationen, etwa bei der Ausführung von Unit Tests, alternative Implementierungen an das abhängige Objekt zu übergeben.

Mit der Nutzung unserer GreetingServiceFactory haben wir also bereits “Dependency Injection” betrieben, weil wir das Wissen über die Abhängigkeit des GreetingService vom LoggingService aus dem GreetingService extrahiert haben. Lassen wir noch einmal den Code für die Ereignisauslösung und -verarbeitung außen vor, so stellt sich der Code wie folgt dar: Dem GreetingService ist die GreetingServiceFactory vorgeschaltet, die sich darum kümmert, die Abhängigkeit des GreetingService vom LoggingService aufzulösen:

```
1  <?php
2  namespace Helloworld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class GreetingServiceFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $greetingService = new GreetingService();
12
13         $greetingService->setLoggingService(
14             $serviceLocator->get('loggingService')
15         );
16
17         return $greetingService;
18     }
19 }
```

Listing 13.1

Der GreetingService greift dann einfach auf den injizierten LoggingService zu, ohne, dass er sich selbst darum gekümmert hätte, dass er denn auch über den LoggingService verfügt. Er verlässt sich dabei darauf, dass eben dieser Service - von wem auch immer - zuvor bereitgestellt wurde, bevor er sich dessen bedient:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  class GreetingService
5  {
6      private $loggingService;
7
8      public function getGreeting()
9      {
10         $this->loggingService->log("getGreeting ausgefuehrt!");
11
12         if(date("H") <= 11)
13             return "Good morning, world!";
14         else if (date("H") > 11 && date("H") < 17)
15             return "Hello, world!";
16         else
17             return "Good evening, world!";
18     }
19
20     public function setLoggingService($loggingService)
21     {
22         return $this->loggingService = $loggingService;
23     }
24
25     public function getLoggingService()
26     {
27         return $this->loggingService;
28     }
29 }
```

Listing 13.2

Der LoggingService verrichtet dann entsprechend noch seine Arbeit:


```
1  <?php
2  namespace HelloWorld\Service;
3
4  class LoggingService
5  {
6      public function log($str)
7      {
8          // hier der Code zu Realisierung des "echten" Loggings
9      }
10 }
```

Listing 13.3

Wenn wir im Rahmen von Unit Tests jetzt vermeiden wollen, dass wir unser Log-File mit vielen “unechten” Zugriffen vollstopfen, können wir dem GreetingService während der Ausführung von Tests einen FakeLoggingService injizieren, der es auf der einen Seite dem GreetingService ermöglicht, klaglos und wie gehabt seine Arbeit zu verrichten und auf der anderen Seite die nutzlosen Logs einfach zu verwerfen:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  class FakeLoggingService
5  {
6      public function log($str)
7      {
8          return;
9      }
10 }
```

Listing 13.4

Im Test können wir dann den FakeLoggingService injizieren:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  class GreetingService extends \PHPUnit_Framework_TestCase
5  {
6      public function testGetGreeting()
7      {
8          $greetingService = new GreetingService();
9          $fakeLoggingService = new FakeLoggingService();
```

```
10         $greetingService->setLoggingService($fakeLoggingService)
11         $result = $greetingService->getGreeting();
12         $greetingSrv = $serviceLocator$this->assertEquals(/* [...] */);
13     }
14 }
```

Listing 13.5

Der Ordnung halber und um sicherzustellen, dass dem GreetingService wirklich immer ein “Logging-artiger Service” zur Verfügung gestellt wird, kann ein Interface eingesetzt werden:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  interface LoggingServiceInterface
5  {
6      public function log($str);
7  }
```

Listing 13.6

Das LoggingServiceInterface würde dann sowohl vom “echten” LoggingService

```
1  <?php
2  namespace HelloWorld\Service;
3
4  class LoggingService implements LoggingServiceInterface
5  {
6      public function log($str)
7      {
8          // hier der Code zu Realisierung des "echten" Loggings
9      }
10 }
```

Listing 13.7

als auch in der “Fake-Implementierung”

```
1  <?php
2  namespace Helloworld\Service;
3
4  class FakeLoggingService implements LoggingServiceInterface
5  {
6      public function log($str)
7      {
8          return;
9      }
10 }
```

Listing 13.8

implementiert. Im GreetingService können wir dann beim entsprechenden “Setter” noch einen Type Hint platzieren, um ein Objekt des richtigen Typs durch den PHP-Interpreter zu erzwingen:

```
1  <?php
2  // [...]
3  public function setLoggingService(LoggingServiceInterface $loggingService)
4  {
5      return $this->loggingService = $loggingService;
6  }
7  // [...]
```

Listing 13.9

Übrigens erschwert die Nutzung der nativen date-Funktion im GreetingService das Unit Testing doch sehr. Im folgenden Kapitel, wenn wir uns mit dem Unit Testing im Detail beschäftigen, werden wir uns für dieses konkrete Beispiel eine mögliche Lösung überlegen.



Zend\Log

Wie wir auch noch sehen werden, brauchen wir uns als Anwendungsentwickler übrigens gar nicht selbst die Mühe machen, eine Logging-Funktionalität zu programmieren: Mit Zend\Log bringt das Zend Framework 2 eine sehr flexible Logging-Implementierung bereits mit. Doch auch dazu später erst mehr.

13.2 Zend\Di für Objektgraphen

Zend\Di kann man sich zunächst am besten als eine generische Factory vorstellen, die man auch als “Dependency Injection Container” bezeichnet. Unsere GreetingServiceFactory war hingegen sehr konkret, weil wir die Abhängigkeit zum LoggingService explizit und händisch aufgelöst haben. Alternativ können wir Zend\Di mit der Auflösung der Abhängigkeit beauftragen:

```
1  <?php
2  namespace Helloworld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class GreetingServiceFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $di = new \Zend\Di\Di();
12
13         $di->configure(new \Zend\Di\Config(array(
14             'definition' => array(
15                 'class' => array(
16                     'Helloworld\Service\GreetingService' => array(
17                         'setLoggingService' => array(
18                             'required' => true
19                         )
20                     )
21                 )
22             ),
23             'instance' => array(
24                 'preferences' => array(
25                     'Helloworld\Service\LoggingServiceInterface'
26                     => 'Helloworld\Service\LoggingService'
27                 )
28             )
29         ));
30
31         $greetingService = $di->get('Helloworld\Service\GreetingService');
32         return $greetingService;
33     }
34 }
35 }
```

Listing 13.10

Bevor wir uns die eigentliche Konfiguration ansehen, zunächst noch ein paar weitere Verbesserungen: Die `GreetingServiceFactory` brauchen wir eigentlich gar nicht mehr, weil `Zend\Di` als generische Implementierung der Factory die Arbeit abnimmt. Stattdessen verlagern wir den Konfigurationscode direkt in die `Module`-Klasse des `HelloWorld`-Moduls. In diesem Zuge machen wir uns auch die Tatsache zu Nutze, dass das Framework uns bereits automatisch eine Instanz von `Zend\Di` als Service im `ServiceManager` zur Verfügung stellt. Diese Instanz können wir auf 2 Arten und Weise erreichen: Entweder, wir fordern bei `ServiceManager` den `DependencyInjector` manuell an:

```
1  <?php
2  // [...]
3  $di = $serviceManager->get('DependencyInjector');
4  // [...]
```

Listing 13.11

und holen darüber den `GreetingService` via

```
1  <?php
2  // [...]
3  $di = $serviceManager->get(`DependencyInjector`);
4  $greetingService = $di->get('HelloWorld\Service\GreetingService');
5  // [...]
```

Listing 13.12

oder aber wir machen es uns noch ein wenig leichter und verwenden den “Fallback-Mechanismus” des `ServiceManager`: Kann der `ServiceManager` nämlich einen angeforderten Service nicht liefern, weil er etwa schlicht nichts von ihm weiß, fragt er noch einmal beim `DependencyInjector` an, ob der vielleicht aushelfen und den entsprechenden Service bereitstellen kann. Und in unserem Fall könnte er. Verwerfen wir also die `GreetingServiceFactory` ganz und verlagern wir den Konfigurationscode für `Zend\Di` direkt in die `module.config.php` des `HelloWorld`-Moduls:

```
1  <?php
2  return array(
3      'di' => array(
4          'definition' => array(
5              'class' => array(
6                  'Helloworld\Service\GreetingService' => array(
7                      'setLoggingService' => array(
8                          'required' => true
9                      )
10                 )
11             )
12         ),
13         'instance' => array(
14             'preferences' => array(
15                 'Helloworld\Service\LoggingServiceInterface'
16                     => 'Helloworld\Service\LoggingService'
17             )
18         )
19     ),
20     'view_manager' => array(
21         'template_path_stack' => array(
22             __DIR__ . '/../view'
23         )
24     ),
25     'router' => array(
26         'routes' => array(
27             'sayhello' => array(
28                 'type' => 'Zend\Mvc\Router\Http\Literal',
29                 'options' => array(
30                     'route' => '/sayhello',
31                     'defaults' => array(
32                         'controller' => 'Helloworld\Controller\Index',
33                         'action' => 'index',
34                     )
35                 )
36             )
37         )
38     ),
39     'controllers' => array(
40         'factories' => array(
41             'Helloworld\Controller\Index'
42                 => 'Helloworld\Controller\IndexControllerFactory'
```

```

43         ),
44         'invokables' => array(
45             'Helloworld\Controller\Widget'
46             => 'Helloworld\Controller\WidgetController'
47         )
48     ),
49     'view_helpers' => array(
50         'invokables' => array(
51             'displayCurrentDate'
52             => 'Helloworld\View\Helper\DisplayCurrentDate'
53         )
54     )
55 );

```

Listing 13.13

Der obere Abschnitt mit dem Schlüssel `di` ist neu und entspricht dem Code, den wir ursprünglich in der `GreetingServiceFactory` hatten. Das Framework schaut nach einem Eintrag unterhalb von `di` und übergibt die Konfiguration, falls vorhanden, an den `DependencyInjector`, der ja automatisch bereitgestellt wird. Die Methode `getServiceConfig()` aus der `Module`-Klasse von `Helloworld` können wir jetzt entfernen. Wir brauchen sie nicht mehr, wenn wir noch eine kleine Anpassung in der `IndexControllerFactory` vornehmen und sicherstellen, dass wir bei der Anforderung des Service den vollqualifizierten Namen `Helloworld\Service\GreetingService` verwenden (bislang hatten wir hier eine Bezeichnung gewählt, die nicht der Klasse entsprach; zur Vermeidung von Namenskollisionen zwischen Services unterschiedlicher Module empfiehlt es sich aber, immer mit vollqualifizierten Klassennamen für die Bezeichnung von Services zu arbeiten):

```

1  <?php
2  namespace Helloworld\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class IndexControllerFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $ctr = new IndexController();
12         $serviceLocator = $serviceLocator->getServiceLocator();
13
14         $greetingSrv = $serviceLocator->get(
15             'Helloworld\Service\GreetingService'
16         );

```

```
17
18         $ctr->setGreetingService($greetingSrv);
19         return $ctr;
20     }
21 }
```

Listing 13.14

Was passiert hier? Der `ServiceManager` kriegt die Anfrage für den `GreetingService` und weil er selbst keine gute Antwort parat hat (die Servicekonfiguration haben wir ja entfernt und alles in den `DependencyInjector` verschoben) konsultiert er nun den `DependencyInjector`, der ihm aus der Patsche hilft, den `GreetingService` instanziert und dabei dafür sorgt, dass auch der `LoggingService` bereitgestellt und dem `GreetingService` verfügbar gemacht wird. Der `ServiceManager` hält dann am Ende eben doch den einsatzbereiten Service in der Hand und gibt ihm freudestrahlend dem Aufrufer zurück.

Noch ein kurzer Hinweis: Es wird einmal vom `ServiceManager` und einmal vom `ServiceLocator` gesprochen, was etwas verwirrend sein kann. Während der `ServiceLocator` ein Interface darstellt, ist der `ServiceManager` dessen konkrete Implementierung, die das Framework direkt mitliefert. Der `ServiceManager` ist also derjenige, der die eigentliche Arbeit übernimmt. Wie an so vielen Stellen im Framework ist der Hintergrund dieser Idee, dass man ja durchaus auf die Idee kommen könnte, den `ServiceManager` gegen eine andere Implementierung auszutauschen. Damit dann der Rest des Frameworks und die darauf aufsetzende Anwendung noch funktioniert, müsste sich diese alternative Implementierung an die Vorgaben des `ServiceLocator`-Interface halten. Wenn also vom `ServiceLocator` die Rede ist, ist eigentlich immer dessen konkrete Standardimplementierung, der `ServiceManager`, gemeint.

Jetzt aber zu den Details der DI-Konfiguration von vorhin: Zunächst wird `Zend\Di` mitgeteilt, dass es da eine Klasse `Helloworld\Service\GreetingService` gibt, die über eine Methode `setLoggingService()` verfügt, über die auf jeden Fall eine Abhängigkeit bereitgestellt werden muss:

```
1  <?php
2  // [...]
3  'definition' => array(
4      'class' => array(
5          'Helloworld\Service\GreetingService' => array(
6              'setLoggingService' => array(
7                  'required' => true
8              )
9          )
10     )
11 )
12 // [...]
```


Listing 13.15

Aber welche nur genau? Hier kommt die RuntimeDefinition ins Spiel: Zend\Di schaut aktiv in die entsprechende Methodendeklaration des GreetingService nach

```
1  <?php
2  // [...]
3  public function setLoggingService(LoggingServiceInterface $loggingService)
4  {
5      return $this->loggingService = $loggingService;
6  }
7  // [...]
```

Listing 13.16

und ermittelt selbstständig die Information darüber, welche Art von Objekt hier zu injizieren ist. Hätten wir hier als Type Hint anstelle des LoggingServiceInterface eine konkrete Klasse angegeben, wären wir schon durch: Zend\Di würde die entsprechende Klasse instanzieren und via setLoggingService() an den GreetingService übergeben, sobald letzterer angefragt wird. Fertig. Allerdings findet Zend\Di in unserem Fall keine konkrete Klasse als Type Hint vor, sondern den Hinweis auf ein Interface. Daher müssen wir hier noch weitere Angaben machen, die Zend\Di zu der tatsächlich zu verwendenden Klasse führen:

```
1  <?php
2  // [...]
3  'instance' => array(
4      'preferences' => array(
5          'Helloworld\Service\LoggingServiceInterface'
6              => 'Helloworld\Service\LoggingService'
7      )
8  )
9  // [...]
```

Listing 13.17

Diese Konfiguration sagt aus: “Wenn du Helloworld\Service\LoggingServiceInterface vorfindest (und nicht weist, was du tun sollst), nutze Helloworld\Service\LoggingService”. Das war schon alles. Jetzt kann der Helloworld\Service\GreetingService indirekt über den ServiceManager angefordert werden:

```
1  <?php
2  // [...]
3  $greetingSrv = $serviceLocator->get('Helloworld\Service\GreetingService');
4  // [...]
```

Listing 13.18

Einige Kapitel früher haben wir auch für den IndexController des Helloworld-Moduls eine eigene Factory, die IndexControllerFactory, erstellt, damit wir dessen Abhängigkeit zum GreetingService auflösen können. Auch dafür können wir bei Bedarf Zend\Di einsetzen, allerdings nur dann, wenn wir den jeweiligen Controller zunächst für das Laden via Zend\Di freigeschaltet haben:

```
1  <?php
2  return array(
3      'di' => array(
4          'allowed_controllers' => array(
5              'helloworld-index-controller'
6          ),
7          'definition' => array(
8              'class' => array(
9                  'Helloworld\Service\GreetingService' => array(
10                     'setLoggingService' => array(
11                         'required' => true
12                     )
13                 ),
14                 'Helloworld\Controller\IndexController' => array(
15                     'setGreetingService' => array(
16                         'required' => true
17                     )
18                 )
19             )
20         ),
21         'instance' => array(
22             'preferences' => array(
23                 'Helloworld\Service\LoggingServiceInterface'
24                     => 'Helloworld\Service\LoggingService'
25             ),
26             'Helloworld\Service\LoggingService' => array(
27                 'parameters' => array(
28                     'logfile' => __DIR__ . '/../data/log.txt'
29                 )
30             ),
31             'alias' => array(
```

```

32         'helloworld-index-controller'
33         => 'Helloworld\Controller\IndexController',
34     ),
35 )
36 ),
37 // [...]
38 );

```

Listing 13.19

Die folgenden Anpassungen an der `module.config.php` sind notwendig, damit auch das Laden des Controllers über `Zend\Di` funktioniert:

- Im Abschnitt `controllers` ist die `IndexControllerFactory` entfernt. Wir wollen sie ja schließlich nicht mehr verwenden, sondern den `IndexController` demnächst über `Zend\Di` erzeugen.
- Der Abschnitt `alias` im Abschnitt `instance` unterhalb von `di` ist neu hinzugekommen. Da bei `Zend\Di` als Alias keine Backslashes erlaubt sind, verwenden wir hier `helloworld-index-controller`. Zuvor hatten wir noch `Helloworld\Controller\Index` verwenden und damit auf die Factory verweisen können. Das geht jetzt leider nicht mehr, macht aber nichts.
- Die Route `sayhello` ist ebenfalls dahingehend angepasst, dass der neue Controller-Alias `helloworld-index-controller` verwendet wird.
- Der Abschnitt `allowed_controllers` ist neu hinzugekommen und nimmt den Wert `helloworld-index-controller` auf, also den Alias des Controllers, den wir zukünftig über `Zend\Di` laden wollen. Wenn wir vergessen, den Controller auf diesem Wege zu "whitelisten", kann er nicht über `Zend\Di` geladen werden, selbst wenn alle sonstigen Konfigurationen korrekt sind - ein Sicherheitsfeature.
- Last but not least müssen wir uns jetzt darum kümmern, dass die Abhängigkeit zum `GreetingService` aufgelöst wird. Die Factory, die sich bislang um die Auflösung der Abhängigkeit zum `GreetingService` gekümmert hat, ist ja nun außer Kraft gesetzt. Um `Zend\Di` diese Aufgabe übernehmen zu lassen, ist im Abschnitt `di > definition > class` ein Eintrag für `Helloworld\Controller\IndexController` hinzugekommen, über den wir `Zend\Di` mitteilen, dass zwingend die Methode `setGreetingService()` des `IndexController` aufgerufen werden muss. Wenn wir nun noch jene Methode des `IndexController` von `Helloworld` mit einem Type Hint ausstatten, kann `Zend\Di` wieder selbst ermitteln, welche Klasse hier injiziert werden muss:

```
1  <?php
2  // [...]
3  public function setGreetingService(
4      \Helloworld\Service\GreetingService $service)
5  {
6      $this->greetingService = $service;
7  }
```

Listing 13.20

Der `ControllerManager` verfügt übrigens über eine eigene `Zend\Di`-Instanz, über die bei Bedarf dann auch eben nur die jeweiligen Controller bezogen werden können.

Via `Zend\Di` konnten wir jetzt also alle unsere Factories eliminieren. Denkt man das weiter, so könnte man im Grunde vollständig von Factories Abstand nehmen und den gesamten Objektgraphen mit all seinen Abhängigkeiten beschreibend aufbauen, ohne andernfalls notwendigen Initialisierungscode zu schreiben.

`Zend\Di` sollte eigentlich der zentrale Dreh- und Angelpunkt auch von `Zend\Mvc` werden und dessen individuelle Factories unnötigt machen. Schaut man sich die Requestverarbeitung durch das Framework an, fällt allerdings auf, dass `Zend\Di` so gut wie gar nicht zum Einsatz kommt und fast alle Services über eigene Factories erzeugt werden. Der `ServiceManager` und die jeweiligen Factories haben an vielen Stellen die Aufgaben von `Zend\Di` übernommen. Die Gründe dafür waren weniger technischer, eher strategischer Natur, will man doch dafür sorgen, dass das Zend Framework 2 auch für Einsteiger leicht erlernbar bleibt. Und `Zend\Di` bringt zwangsläufig eine ganze Menge versteckter “Magie” in die Sache. Ähnlich verhält es sich im Grunde auch mit der eigenen Entscheidung für bzw. gegen `Zend\Di` und Factories. Beide Ansätze sind probate Mittel, um eine lose Kopplung der einzelnen Akteure zu erreichen und damit ein System zu entwickeln, dass auch nachhaltig test-, wart- und erweiterbar bleibt. Der Anwendungsentwickler hat die freie Wahl. Durch den oben beschriebenen “Fallback”-Mechanismus lassen sich beide Wege ja auch problemlos miteinander kombinieren und sich so je nach Situation das beste Vorgehen wählen.

Alles irgendwie viel zu schwierig? Keine Sorge - es sieht viel schlimmer aus, als es tatsächlich ist. Es braucht nur ein wenig Übung.



Alternative Ansätze für die “Definition”

Neben der hier gezeigten `RuntimeDefinition`, die den Reflection-Mechanismus verwendet, um die Struktur des Codes für die Auflösung von Abhängigkeiten zu verstehen, können auch andere Varianten, wie etwa die `CompilerDefinition` eingesetzt werden, die in komplexen Szenarien ggf. Geschwindigkeitsvorteile zu Lasten der Bequemlichkeit bringen kann.

13.3 Zend\Di für das Konfigurationsmanagement

“Dependency Injection” im Allgemeinen und Zend\Di im Speziellen ist im Kern für zwei Dinge hilfreich: Zum einen lassen sich, wie oben exemplarisch skizziert, komplexe Objektgraphen zur Laufzeit “zusammenstecken”, ohne dass die abhängigen Objekte selbst einen Finger dafür krumm machen müssten oder die hartverdrahteten Abhängigkeiten etwa beim Unit Testing in irgendeiner Form im Weg stünden. Zum anderen lassen sich einzelne Objekte auch jenseits ihrer Abhängigkeit mit Hilfe von “Dependency Injection” konfigurieren. So werden für den Zugriff auf eine Datenbank oder einen externen Webservice an irgendeiner Stelle im verantwortlichen Objekt die Informationen zum Host sowie etwaige Zugangsdaten benötigt. Zwar könnte der jeweilige Service unserer Anwendung jetzt selbst aktiv über den ServiceManager auf die Konfiguration der Anwendung zugreifen und sich dort entsprechend die passenden Werte herausfischen

```
1  <?php
2  // [...]
3  $config = $serviceManager->get(`Config`);
4  $host = $config['dbConfig']['host'];
5  $user = $config['dbConfig']['user'];
6  $pwd = $config['dbConfig']['pwd'];
7  // [...]
```

Listing 13.21

allerdings wären dort dann die Abhängigkeiten sowohl zum ServiceManager als auch das Wissen über die interne Struktur der Konfiguration fest verdrahtet. Äußert ungünstig. Ein anderes Beispiel: Wie teilen wir unserem LoggingService am besten mit, in welches File er loggen soll? Am besten erweitert wir dazu den Konstruktor des Services, so dass wir die Konfiguration von Außen injizieren können:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  class LoggingService implements LoggingServiceInterface
5  {
6      private $logfile = null;
7
8      public function __construct($logfile)
9      {
10         $this->logfile = $logfile;
11     }
12
13     public function log($str)
```

```

14     {
15         file_put_contents($this->logfile, $str, FILE_APPEND);
16     }
17 }

```

Listing 13.22

Zusätzlich erweitern wir die DI-Konfiguration in der `module.config.php`:

```

1  <?php
2  // [...]
3  'di' => array(
4      'definition' => array(
5          'class' => array(
6              'Helloworld\Service\GreetingService' => array(
7                  'setLoggingService' => array(
8                      'required' => true
9                  )
10             )
11         )
12     ),
13     'instance' => array(
14         'preferences' => array(
15             'Helloworld\Service\LoggingServiceInterface'
16                 => 'Helloworld\Service\LoggingService'
17         ),
18         'Helloworld\Service\LoggingService' => array(
19             'parameters' => array(
20                 'logfile' => __DIR__ . '/../.../data/log.txt'
21             )
22         )
23     )
24 )
25 // [...]

```

Listing 13.23

Das Schöne daran ist, dass wir die Konfiguration des Logfiles an “Ort und Stelle” vornehmen können, also physisch bei der entsprechenden Service-Konfiguration, ohne sie aber direkt innerhalb der Service-Klasse selbst hart verdrahten zu müssen. Im Gegensatz zu einer endlosen Konfigurationsdatei, mit zig zusammenhangslosen Konfigurationswerten aneinandergereiht, wie dies etwa in Version 1 des Frameworks mit der `application.ini` noch der Fall war, weiß man so direkt, an welcher Stelle man suchen muss, wenn es mal etwas zu ändern gilt.

14 Persistenz mit Zend\Db

Persistenz, also das langfristige Speichern von Daten, etwa in Datenbanken, wird seit langer Zeit von PHP bereits gut unterstützt. Für die PDO-Erweiterung etwa gibt es datenbankspezifische Treiber für die gängigsten Hersteller und Systeme, so dass man es mit PHP nahezu immer mühelos schafft, eine Datenbank anzubinden und Daten zu persistieren. Mit Zend\Db hat das Zend Framework 2 zusätzliche Unterstützung für die Arbeit mit Datenbanken im Gepäck und macht Persistenz damit noch ein Stück einfacher.

Zend\Db ist übrigens kein [ORM](#)¹-System und steht damit nicht in Konkurrenz etwa zu [Doctrine](#)² oder [Propel](#)³, sondern kann als “leichtgewichtige” Alternative, als zusätzliche Abstraktion verstanden und eingesetzt werden. Wer “Doctrine” oder ein anderes ORM-System einsetzt, wird also wohl auf Zend\Db verzichten. Das Thema “Persistenz” ist leider derart mit unterschiedlichen, teils konträren Bezeichnungen und Definitionen überladen, dass es wirklich schwer fällt, zu verstehen, was sich eigentlich genau hinter einer Implementierung verbirgt. Um Zend\Db besser einordnen zu können - auch auf die Gefahr hin, dass die folgenden Ausführungen von dem ein oder anderen als zu starke Vereinfachung kritisiert werden könnten - schauen wir uns kurz die wichtigsten Ansätze für “Persistenz” an:

- **Objekt Relational Mapper (ORM):** Ausgangspunkt für ein ORM-System ist die Denkweise, dass wir zwar objektorientiert programmieren, die Daten dann aber relational speichern. Zwar sind einige Parallelen zwischen den beiden Paradigmen zu finden, allerdings gibt es auch handfeste Unterschiede. Wenn man so will, versucht ein ORM-System daher, die Datenbank, sozusagen die “artfremde Welt”, für die objektorientiert denkenden und handelnden Anwendungsentwickler möglichst unsichtbar zu machen: Man erzeugt und manipuliert Objekte und das ORM-System sorgt für alles Weitere in Richtung Datenbank. Häufig kommen ORM-Systeme auch mit SQL-Alternativen daher, so hat “Doctrine 2” etwa die DQL (Doctrine Query Language), die sehr nach SQL aussieht (was für geübte “SQLer” auf jeden Fall auch ein Vorteil ist), aber einige zusätzliche Eigenschaften hat, um die Objektorientierung bestmöglich aufzugreifen und Datenbankoperationen dadurch zu vereinfachen.
- **Data Mapper:** Ein “Data Mapper” wird häufig mit einem ORM-System gleichgesetzt, kann aber auch vollkommen andere “Backends” unterstützen, als relationale Datenbanken. So bietet “Doctrine 2” etwa auch die Möglichkeit, Persistenz in “MongoDB” oder “CouchDB”, beides dokumentenorientierte Systeme, zu realisieren. Wenn man von ORM und “Doctrine 2” spricht, müsste man korrekterweise von “Doctrine 2 ORM” sprechen, gibt es ja eben doch auch “Doctrine 2 ODM” (Object Document Mapper). Das Hauptcharakteristikum eines Data Mappers ist die Tatsache, dass es die eine Welt (z.B. die objektorientierte) vollkommen

¹http://de.wikipedia.org/wiki/Objektrelationale_Abbildung

²<http://www.doctrine-project.org/>

³<http://www.propelorm.org/>

von der anderen Welt (z.B. die relationale) abkoppelt und er sich als zwischengeschalteter “Transformer” nützlich macht. Eine wichtige Eigenschaft eines “Data Mapper” ist auch die Tatsache, dass er Objekte persistieren oder laden kann, bei denen er sich (gleichzeitig) mehrerer unterschiedlicher Quellen, etwa mehrerer Datenbanktabellen, bedient.

- **Table Data Gateway:** Während ein “Data Mapper” bzw. ein ORM-System konzeptionell versucht, Informationen über die detaillierten Datenstrukturen “der anderen Welt” so weit wie möglich zu verbergen, steht hinter dem “Table Data Gateway” ein ganz anderer Ansatz. Hier wird mit offenen Karten gespielt und für jede Tabelle in der Datenbank ein Objekt erzeugt, dass sich um die Operation rund um diese Tabelle kümmert. Wichtig ist hier, im Gegensatz zum “Data Mapper”, der Fokus auf eine Tabelle in einer relationalen Datenbank.
- **Row Data Gateway:** Während “Table Data Gateway” eine Tabelle durch ein Objekt repräsentiert, steht ein “Row Data Gateway” als Objekt für eine Zeile, einen “Eintrag”, wenn man so will, einer Tabelle in der Datenbank. Eine wichtige Eigenschaft des “Row Data Gateway” ist die Tatsache, dass das jeweilige Objekt - im Gegensatz zum “Data Mapper” - den notwendigen Code mit sich führt, um sich selbst zu laden oder zu persistieren. Sie sind dahingehend also Autonom. Bei einem “Data Mapper” sind hingegen die aus der Datenbank geladenen Objekte, in diesem Kontext häufig auch als “Entities” bezeichnet, dazu nicht in der Lage. Stattdessen wird ein zentraler “Entity Manager” bereitgestellt, der sich um die Persistenz der Objekte kümmert. Ein Vorteil des “Entity Manager” ist die Tatsache, dass er im Rahmen einer “Unit of Work” operieren und damit mehrere, ähnliche SQL-Statements zu einem einzigen Statement zusammenfassen kann.
- **Active Record:** Ein “Active Record” ist zu vergleichen mit dem “Row Data Gateway”. Der einzige, tatsächlich auch eher definitorische Unterschied, ist, dass sich ein “Active Record” im Gegensatz zu einem “Row Data Gateway” neben den Belangen der Persistenz auch um fachliche Angelegenheiten kümmern darf, also auch Code enthält, der sich nicht nur um die Abbildung eines Objektes auf eine Tabellenzeile kümmert, sondern sich auch mit der “Fachlichkeit” der Anwendung beschäftigt.
- **Data Access Object (DAO):** Schwer definitorisch zu greifen ist das “Data Access Object”. Nimmt man die Definition aus dem [Core J2EE Pattern Catalog](http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html)⁴, so stellt es den Zugriff auf eine Datenquelle her, baut also etwa die Verbindung zu einer Datenbank auf, wäre demnach sehr “low level” und könnte als Basis für die zuvor genannten Konstrukte dienen, die sich ihrerseits darum nämlich definitorisch auch gar nicht kümmern. Nun ist J2E (früher: J2EE) ja gedanklich immer etwas “breiter” aufgestellt und es ist schnell die Rede davon, dass bestimmte Objekttypen (User, Product, Catagory, etc.) ja auch unterschiedlichen Quellen bezogen werden könnten, also etwa aus Datenbanksystemen, aber auch aus Flatfiles, einer LDAP-Implementierung oder ähnlichem. Je nach Datenquelle gäbe es dort dann ein jeweils passendes “Data Access Object”. Da sich oftmals aber alle persistierten Objekte im identischen Storage befinden, spielen “Data Access Objects” keine so große Rolle, bzw. sind eher im Hintergrund. Außer aber jemand meint mit “Data Access Object” eigentlich das “Table Data Gateway”. Diese Begriffe werden leider nämlich häufig auch synonym verwendet.

⁴<http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>

14.1 Datenbanken anbinden

Der `Zend\Db\Adapter` hat die Aufgabe, die Verbindung zur Datenbank herzustellen und die Eigenheiten der SQL-Implementierung einzelner Systeme, sowie die vielen unterschiedlichen Arten der Datenbankanbindung seitens PHP, zu vereinheitlichen. Denn zum einen unterstützen nicht alle Anbieter mit ihren Systemen alle SQL-Standards, bzw. bieten häufig darüber hinausgehende, proprietäre Erweiterungen an und zum anderen verfügt PHP selbst über eine ganze Reihe verschiedener Möglichkeiten für die Datenbankinteraktion, neben PDO ja etwa auch noch MySQLi und das ältere MySQL (die Erweiterung, nicht das DBMS). Wenn man also `Zend\Db\Adapter` anstelle der nativen Funktionen und Objekte einsetzt, erhöht man die Portabilität. Wenn wir noch einmal auf die Definitionen von oben zurückkommen, könnten wir `Zend\Db\Adapter` als unser “Data Access Object” bezeichnen.

Damit `Zend\Db` ordnungsgemäß funktioniert, muss zunächst die Datenbankverbindung über den Adapter hergestellt werden:

```
1  <?php
2  $adapter = new \Zend\Db\Adapter\Adapter(
3      array(
4          'driver' => 'Pdo_Mysql',
5          'hostname' => 'localhost'
6          'database' => 'app',
7          'username' => 'root',
8          'password' => ''
9      )
10 );
```

Listing 14.1



Verwendung von Passwörtern

In einem Produktivsystem sorgen wir natürlich dafür, dass wir starke Passwörter einsetzen und besser noch überhaupt nicht mit dem User “root” arbeiten, sondern mit speziellen Nutzern, die wir in ihren Möglichkeiten auf das Nötigste beschränken können.

Der MySQL-Dienst wurde im Vorfeld auf dem `localhost` verfügbar gemacht, die Datenbank `app` angelegt und dem Nutzer `root` ohne Passwort zugänglich gemacht (in einem Entwicklungssystem so sicherlich okay). Zusätzlich wurde die Tabelle `log` mit Hilfe des folgenden SQL-Statements angelegt:

```
1 CREATE TABLE log (  
2     id int(10) NOT NULL auto_increment,  
3     ip varchar(16) NOT NULL,  
4     timestamp varchar(10) NOT NULL,  
5     PRIMARY KEY (id)  
6 );
```

14.2 SQL-Statements erzeugen & ausführen

Hat man der Tabelle manuell einige Datensätze hinzugefügt, lassen sich diese über den zuvor erzeugten Adapter abrufen:

```
1 <?php  
2 $stmt = $adapter->createStatement('SELECT * FROM log');  
3 $results = $stmt->execute();  
4  
5 foreach($results as $result)  
6     var_dump($result);
```

Listing 14.2

Eine sehr elegante, objektorientierte Art und Weise, SQL-Statements zu erzeugen, ist Zend\Db\Sql:

```
1 <?php  
2 $sql = new \Zend\Db\Sql\Sql($adapter);  
3 $select = $sql->select();  
4 $select->from('log');  
5 $statement = $sql->prepareStatementForSqlObject($select);  
6 $results = $statement->execute();
```

Listing 14.3

Auch eine WHERE-Clause lässt sich einfach hinzufügen:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $select = $sql->select();
4 $select->from('log');
5 $select->where(array('ip' => '127.0.0.1'));
6 $statement = $sql->prepareStatementForSqlObject($select);
7 $results = $statement->execute();
```

Listing 14.4

Gleiches gilt für die anderen üblichen Konstrukte wie LIMIT, OFFSET, ORDER, etc., deren Nutzung im Grunde selbsterklärend ist. Auch ein JOIN ist grundsätzlich simpel:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $select = $sql->select();
4 $select->from('log');
5 $select->join('host', 'host.ip = log.ip');
6 $select->where(array('log.ip' => '127.0.0.1'));
7 $statement = $sql->prepareStatementForSqlObject($select);
8 $results = $statement->execute();
```

Listing 14.5

In diesem Fall wird die zusätzliche Tabelle host referenziert, die mit dem folgenden Statement erzeugt wurde:

```
1 CREATE TABLE host (
2     id int(10) NOT NULL auto_increment,
3     ip varchar(16) NOT NULL,
4     hostname varchar(100) NOT NULL,
5     PRIMARY KEY (id)
6 );
```

Bei diesem “Join” handelt es sich um einen “Inner Join”. Andere Arten von “Joins” werden ebenso unterstützt:

```
1  <?php
2  $sql = new \Zend\Db\Sql\Sql($adapter);
3  $select = $sql->select();
4  $select->from('log');
5
6  $select->join('host',
7      'host.ip = log.ip',
8      array('*'),
9      \Zend\Db\Sql\Select::JOIN_LEFT
10 );
11
12 $select->where(array('log.ip' => '127.0.0.1'));
13 $statement = $sql->prepareStatementForSqlObject($select);
14 $results = $statement->execute();
```

Listing 14.6

Der Parameter `array('*')` deutet in diesem Fall auf die Spalten hin, die im Rahmen des “Joins” von der Tabelle `host` in das Ergebnis übernommen werden sollen. Mit `array('*')` werden alle Spalten verwendet, unter Angabe konkreter Spalten kann das Ergebnis beschränkt werden, ein assoziatives Array kann zum Einsatz von Alias-Werten für die zu berücksichtigten Spalten verwendet werden.

Auf diese Art und Weise können Daten auch elegant in die Datenbank geschrieben

```
1  <?php
2  $sql = new \Zend\Db\Sql\Sql($adapter);
3  $insert = $sql->insert('host');
4  $insert->columns(array('ip', 'hostname'));
5
6  $insert->values(array(
7      'ip' => '192.168.1.15',
8      'hostname' => 'michaels-ipad')
9  );
10
11 $statement = $sql->prepareStatementForSqlObject($insert);
12 $results = $statement->execute();
```

Listing 14.7

aktualisiert

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $update = $sql->update('host');
4 $update->set(array('ip' => '192.168.1.20'));
5 $update->where('hostname = "michaels-ipad"');
6 $statement = $sql->prepareStatementForSqlObject($update);
7 $results = $statement->execute();
```

Listing 14.8

und gelöscht werden:

```
1 <?php
2 $sql = new \Zend\Db\Sql\Sql($adapter);
3 $delete = $sql->delete('');
4 $delete->from('host');
5 $delete->where('hostname = "michaels-ipad"');
6 $statement = $sql->prepareStatementForSqlObject($delete);
7 $results = $statement->execute();
```

Listing 14.9



Vielfältige Möglichkeiten

Weitere Details zu `Zend\Db\Sql\Sql` finden sich bei Bedarf in der [offiziellen Dokumentation](http://packages.zendframework.com/docs/latest/manual/en/modules/zend.db.sql.html)⁵.

14.3 Mit Tabellen und Entities arbeiten

Noch etwas leichter kann man sich das Leben machen, wenn man `Zend\Db\TableGateway` verwendet. Wie in der Einleitung zu diesem Kapitel beschrieben, repräsentiert ein `TableGateway`-Objekt eine Tabelle in der Datenbank. Eine Abfrage lässt sich wie folgt realisieren:

⁵<http://packages.zendframework.com/docs/latest/manual/en/modules/zend.db.sql.html>

```
1  <?php
2  $hostTable = new \Zend\Db\TableGateway\TableGateway('host', $adapter);
3  $results = $hostTable->select(array('hostname' => 'michaels-mac'));
4
5  foreach ($results as $result)
6      var_dump($result);
```

Listing 14.10

Auch hier wurde selbstverständlich zuvor der entsprechende Adapter erzeugt:

```
1  <?php
2  $adapter = new \Zend\Db\Adapter\Adapter(
3      array(
4          'driver' => 'Pdo_Mysql',
5          'database' => 'app',
6          'username' => 'root',
7          'password' => ''
8      )
9  );
```

Listing 14.11

Wenn man mit den zurückgelieferten Daten noch weiter arbeiten will, etwa Datensätze bearbeiten oder löschen, können auch Row Data Gateway-Objekte angefordert werden, die einen Datensatz in der jeweiligen Tabelle repräsentieren und Funktionen mitbringen, um sie zu manipulieren. Dazu muss die Abfragen von oben wie folgt modifiziert werden:

```
1  <?php
2  $hostTable = new \Zend\Db\TableGateway\TableGateway(
3      'host',
4      $adapter,
5      new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
6  );
7
8  $results = $hostTable->select(array('hostname' => 'michaels-mac'));
9
10 foreach ($results as $result)
11     var_dump($result);
```

Listing 14.12

Bei der Erzeugung des TableGateway übergeben wir als zusätzlichen Parameter das RowGatewayFeature, was dazu führt, dass das TableGateway die einzelnen Ergebnisse der Abfrage nicht mehr als Arrays

oder `ArrayObjects` bereitstellt, sondern eben als `Collection` von `RowGateway`-Objekten. Diese bringen eine `save()` und `delete()`-Methode mit, um Änderungen an dem Datensatz in die Datenbank zurückzuschreiben, oder eben um ihn zu löschen:

```
1  <?php
2  $hostTable = new \Zend\Db\TableGateway\TableGateway(
3      'host',
4      $adapter,
5      new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
6  );
7
8  $results = $hostTable->select(array('ip' => '127.0.0.1'));
9  $result = $results->current();
10 $result->hostname = 'michaels-macbook';
11 $result->save(); // oder: $result->delete();
```

Listing 14.13

Die `Row Data Gateway`-Objekte, die von einer Abfrage zurückgegeben werden, sind alle vom Typ `Zend\Db\RowGateway\RowGateway`. Sie bieten damit den Mehrwert, dass sie sich um ihre eigene Persistenz kümmern können, tragen dafür aber keine fachliche Information. Hier kommt die sogenannte “Hydration” ins Spiel, bei der die aus der Datenbank geladenen Daten transparent in ein fachliches Objekt überführt werden. Dabei verliert das Objekt zwar seine Persistenz-Funktionen, aber bei lesenden Operationen kann man darauf zunächst ja gut verzichten.

Zunächst legen wir dazu das fachliche Objekt, eine sog. “Entity” im `Helloworld`-Modul unter `/src/Helloworld/Entity/Host.php` an:

```
1  <?php
2  namespace Helloworld\Entity;
3
4  class Host
5  {
6      protected $ip;
7      protected $hostname;
8
9      public function getHostname()
10     {
11         return $this->hostname;
12     }
13
14     public function getIp()
15     {
```

```
16         return $this->ip;
17     }
18 }
```

Listing 14.14

Der Host hat zunächst nur zwei protected-Eigenschaften und “Getter”. Die folgende Abfrage führt nun dazu, dass in der Variable `hosts` Objekte vom Typ `host` vorgehalten werden:

```
1  <?php
2  $hostTable = new \Zend\Db\TableGateway\TableGateway(
3      'host',
4      $adapter,
5      new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
6  );
7
8  $results = $hostTable->select(array('ip' => '127.0.0.1'));
9
10 $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
11     new \Zend\Stdlib\Hydrator\Reflection(),
12     new \Helloworld\Entity\Host()
13 );
14
15 $hosts->initialize($results->toArray());
```

Listing 14.15

Das Abfrage-Ergebnis wird in ein `HydratingResultSet` überführt. Dazu muss eine Aussage dazu getroffen werden, wie die Daten zugeordnet werden sollen (`Reflection`) und wo sie letztendlich landen soll (`Host`). Dafür kommt das sog. “Prototype Pattern” zum Einsatz, bei dem ein exemplarisches Objekt, in diesem Fall die neu erzeugte Instanz der `Host`-Klasse, für jeden Datensatz im `ResultSet` der Abfrage dupliziert (`clone`) und jeweils mit den Daten ausgestattet wird. Man steckt also ein “prototypisches” Objekt rein und bekommt so viele, mit den richtigen Daten initialisierte Klone dieses Objektes zurück, wie erforderlich.

Warum aber werden die notwendigen Objekte nicht einfach über den `new`-Operator bei Bedarf instanziiert? Die Idee dahinter ist, dass das Objekt neben den Datenfeldern, die gefüllt werden sollen, auch noch über weitere Abhängigkeiten zu anderen Objekten verfügen kann, die nicht automatisch und problemlos aufgelöst werden könnten. Stattdessen wird ein bereits fertig konfiguriertes Objekt schlichtweg geklont und das Problem damit umgangen.

Die Annahme, die hier getroffen wird ist, dass die Benennung der Tabellenspalten mit denen der Objekteigenschaften übereinstimmen. Ist dies nicht (immer) der Fall, so bekommt man ggf. nur teilgefüllte Objekte zurück:


```
1 object(Helloworld\Entity\Host)#236 (2) {
2     ["ip":protected]=> string(9) "127.0.0.1"
3     ["hostname":protected]=> NULL
4 }
```

Der hostname ist leer, weil das entsprechende Datenbankfeld mit workstation benannt ist. Jetzt könnte man entweder das Datenbankfeld zu “hostname” umbenennen oder die Objekteigenschaft zu “workstation”. Da dies aber nicht immer sinnvoll oder möglich ist, kann man sich stattdessen mit einem eigenen, abgeleiteten “Hydrator” behelfen, der sich um das notwendige “Mapping” kümmert:

```
1 <?php
2 namespace Helloworld\Mapper;
3
4 use Zend\Stdlib\Hydrator\Reflection;
5 use Helloworld\Entity\Host;
6
7 class HostHydrator extends Reflection
8 {
9     public function hydrate(array $data, $object)
10    {
11        if (!$object instanceof Host) {
12            throw new \InvalidArgumentException(
13                '$object must be an instance of Helloworld\Entity\Host'
14            );
15        }
16
17        $data = $this->mapField('workstation', 'hostname', $data);
18        return parent::hydrate($data, $object);
19    }
20
21    protected function mapField($keyFrom, $keyTo, array $array)
22    {
23        $array[$keyTo] = $array[$keyFrom];
24        unset($array[$keyFrom]);
25        return $array;
26    }
27 }
```

Listing 14.16

Diese Klasse liegt im Verzeichnis src/Helloworld/Mapper/HostHydrator.php und bildet entsprechend das Feld workstation auf das Feld hostname ab. Jetzt muss man nur noch den Aufruf so modifizieren, dass der passende “Hydrator” Anwendung findet:

```
1  <?php
2  $hostTable = new \Zend\Db\TableGateway\TableGateway(
3      'host',
4      $adapter,
5      new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
6  );
7
8  $results = $hostTable->select(array('ip' => '127.0.0.1'));
9
10 $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
11     new \Helloworld\Mapper\HostHydrator(),
12     new \Helloworld\Entity\Host()
13 );
14
15 $hosts->initialize($results->toArray());
16 var_dump($hosts->current());
```

Listing 14.17

Nun funktioniert das Laden der Daten wieder wie gewünscht:

```
1  object(Helloworld\Entity\Host)#236 (2) {
2      ["ip":protected]=> string(9) "127.0.0.1"
3      ["hostname":protected]=> string(12) "michaels-mac"
4  }
```

14.4 Organisation von Datenbankabfragen

Bislang haben wir die Datenbankverbindung und die Datenbankabfragen direkt in einem Controller hergestellt bzw. ausgeführt. Zu Demonstrationszwecken war das sicherlich legitim. In einer echten Anwendung dagegen benötigt man eine bessere Art und Weise, um mit Datenbankabfragen umzugehen. Es empfiehlt sich das folgende Vorgehen: Die Erzeugung des Datenbankadapters, der den Zugriff auf die Datenbank ermöglicht, wird in den ServiceManager verschoben, die Verbindungsdaten in einer Konfigurationsdatei abgelegt und die Datenabfragen rund um einzelne Tabellen bzw. Entites in speziellen Objekten gekapselt.

Datenbankadapter über ServiceManager erzeugen

Um den ServiceManager für die Erzeugung des Datenbankadapters vorzubereiten ändern wir zunächst die `module.config.php` des Helloworld-Moduls wie folgt:

```
1  <?php
2  // [...]
3  'service_manager' => array(
4      'factories' => array(
5          'Zend\Db\Adapter\Adapter' => function ($sm) {
6              $config = $sm->get('Config');
7              $dbParams = $config['dbParams'];
8
9              return new Zend\Db\Adapter\Adapter(array(
10                 'driver'      => 'pdo',
11                 'dsn'         =>
12                     'mysql:dbname=' . $dbParams['database']
13                     . ';host=' . $dbParams['hostname'],
14                 'database'   => $dbParams['database'],
15                 'username'   => $dbParams['username'],
16                 'password'   => $dbParams['password'],
17                 'hostname'   => $dbParams['hostname'],
18             ));
19         },
20     ),
21 )
22 // [...]
```

Listing 14.18

Etwaige, bereits existierende Service-Definitionen sollten ggf. natürlich erhalten bleiben. Es ist übrigens grundsätzlich frei wählbar, in welchem Modul diese Service-Definition ihren Platz findet. Wir können sie wie oben gezeigt im `HelloWorld`-Modul unterbringen, aber ggf. auch in `Application`. Will man den Datenbankadapter über mehrere “Funktionsmodule” hinweg verwenden, so bietet es sich an, die Definition in das `Application`-Modul zu verlagern, schlicht deshalb, weil man so “per Konvention” weiß, wo man schauen muss, wenn man die Definition für einen Service sucht, der über mehrere Module hinweg verwendet wird.

Die Callback-Funktion von oben erzeugt den Datenbankadapter und greift dazu auf Verbindungsdaten zu, die in einer Konfigurationsdatei hinterlegt sind. Ich habe dazu die Datei `dev1.local.php` verwendet:

```
1 <?php
2 return array(
3     'dbParams' => array(
4         'database' => 'app',
5         'username' => 'root',
6         'password' => '',
7         'hostname' => 'localhost',
8     )
9 );
```

Listing 14.19

So können wir nun an der Stelle, an der der Datenbankadapter benötigt wird, den ServiceManager mit der Erzeugung beauftragen. Aus

```
1 <?php
2 $adapter = new \Zend\Db\Adapter\Adapter(
3     array(
4         'driver' => 'Pdo_Mysql',
5         'database' => 'app',
6         'username' => 'root',
7         'password' => ''
8     )
9 );
```

Listing 14.20

etwa in einem Controller wird daher

```
1 <?php
2 $adapter = $this->getServiceLocator()->get('Zend\Db\Adapter\Adapter');
```

Listing 14.21

Kapseln gleichartiger Abfragen

Um zu vermeiden, dass Datenbankabfragen über die gesamte Anwendung verteilt und Änderungen an Datenschemata oder den Abfragen selbst zu einem Wartungsalbtraum werden, empfiehlt es sich, von vornherein alle Abfragen, die sich um die gleiche “Entity” bzw. Datenbanktabelle drehen, an einem Ort zu konsolidieren. Dazu lässt sich prima das TableGateway nutzen, dass wir bereits kennengelernt haben und das den einfachen Zugriff auf Datenbanktabellen ermöglicht. Für jede Entität legen wir ein eigenes, spezifisches TableGateway an. Die Bezeichnung dafür lehnen wir an die jeweilige “Entity” an:

```
1  <?php
2  namespace Helloworld\Mapper;
3
4  use Helloworld\Entity\Host as HostEntity;
5  use Zend\Stdlib\Hydrator\HydratorInterface;
6  use Zend\Db\TableGateway\TableGateway;
7  use Zend\Db\TableGateway\Feature\RowGatewayFeature;
8
9  class Host extends TableGateway
10 {
11     protected $tableName = 'host';
12     protected $idCol = 'id';
13     protected $entityPrototype = null;
14     protected $hydrator = null;
15
16     public function __construct($adapter)
17     {
18         parent::__construct($this->tableName,
19                             $adapter,
20                             new RowGatewayFeature($this->idCol)
21                             );
22
23         $this->entityPrototype = new HostEntity();
24         $this->hydrator = new HostHydrator();
25     }
26
27     public function findByIp($ip)
28     {
29         return $this->hydrate(
30             $this->select(array('ip' => $ip))
31             );
32     }
33
34     public function hydrate($results)
35     {
36         $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
37             $this->hydrator,
38             $this->entityPrototype
39             );
40
41         return $hosts->initialize($results->toArray());
42     }
```

43 }

Listing 14.22

Den Code für den Mapper legen wir in `src/Helloworld/Mapper/Host.php` ab und damit im gleichen Verzeichnis, in dem auch bereits unser `HostHydrator` liegt und den wir uns auch hier wieder zu nutze machen.

Natürlich lässt sich dieser Code noch weiter optimieren, keine Frage. Etwa wäre es sinnvoll, alle Konfigurationswerte und Abhängigkeiten zu injizieren, als dort fest zu verdrahten. Aber für dieses Beispiel soll es so mal ausreichen. Wenn man es sich noch einfacher machen will, greift man gleich auf den fertigen [AbstractDbMapper](https://github.com/ZF-Commons/ZfcBase/blob/master/src/ZfcBase/Mapper/AbstractDbMapper.php)⁶ des [ZF-Commons](https://github.com/ZF-Commons)⁷ git-Repositories zurück. Dort wurde diese Arbeit, quasi von “offizieller Stelle” (das Repository wird von Entwicklern gepflegt, die auch am Framework selbst beteiligt sind), bereits erledigt. Ein Blick lohnt sich auf jeden Fall.

Der nun im Controller noch notwendige Aufruf ist bereits sehr kompakt:

```
1 <?php
2 $adapter = $this->getServiceLocator()->get('Zend\Db\Adapter\Adapter');
3 $mapper = new \Helloworld\Mapper\Host($adapter);
4 $hosts = $mapper->findByIp('127.0.0.1');
```

Listing 14.23

Wir könnten den Code noch weiter kürzen, wenn wir den Adapter automatisch bei der Erzeugung des “Mappers” injizieren:

```
1 <?php
2 // [...]
3 'service_manager' => array(
4     'factories' => array(
5         'Zend\Db\Adapter\Adapter' => function ($sm) {
6             $config = $sm->get('Config');
7             $dbParams = $config['dbParams'];
8
9             return new Zend\Db\Adapter\Adapter(array(
10                 'driver'      => 'pdo',
11                 'dsn'         =>
12                     'mysql:dbname=' . $dbParams['database']
13                     . ';host=' . $dbParams['hostname'],
14                 'database'    => $dbParams['database'],
15                 'username'    => $dbParams['username'],
```

⁶<https://github.com/ZF-Commons/ZfcBase/blob/master/src/ZfcBase/Mapper/AbstractDbMapper.php>

⁷<https://github.com/ZF-Commons>

```

16         'password' => $dbParams['password'],
17         'hostname' => $dbParams['hostname'],
18     ));
19 },
20 'Helloworld\Mapper\Host' => function ($sm) {
21     return new \Helloworld\Mapper\Host(
22         $sm->get('Zend\Db\Adapter\Adapter')
23     );
24 }
25 ),
26 ),
27 // [...]

```

Listing 14.24

Nun ist der im Controller noch notwendige Aufruf ein 1-Zeiler:

```

1  <?php
2  $hosts = $this->getServiceLocator()
3      ->get('Helloworld\Mapper\Host')->findByIp('127.0.0.1');

```

Listing 14.25

Unter Einsatz des TableGateway können wir nun auch elegant das Problem lösen, dass die “Entities” durch die “Hydration” ihre Funktionen zur Persistenz verlieren. Eine Änderung an einer “Entity” lässt sich nun ja nicht mehr via save() direkt in die Datenbank schreiben. Damit beauftragen wir nun ebenfalls den Host-Mapper:

```

1  <?php
2
3  namespace Helloworld\Mapper;
4
5  use Helloworld\Entity\Host as HostEntity;
6  use Zend\Stdlib\Hydrator\HydratorInterface;
7  use Zend\Db\TableGateway\TableGateway;
8  use Zend\Db\TableGateway\Feature\RowGatewayFeature;
9  use Zend\Db\Sql\Sql;
10 use Zend\Db\Sql\Insert;
11
12 class Host extends TableGateway
13 {
14     protected $tableName = 'host';
15     protected $idCol = 'id';

```

```
16     protected $entityPrototype = null;
17     protected $hydrator = null;
18
19     public function __construct($adapter)
20     {
21         parent::__construct($this->tableName,
22             $adapter,
23             new RowGatewayFeature($this->idCol)
24         );
25
26         $this->entityPrototype = new HostEntity();
27         $this->hydrator = new HostHydrator();
28     }
29
30     public function findByIp($ip)
31     {
32         return $this->hydrate(
33             $this->select(array('ip' => $ip))
34         );
35     }
36
37     public function hydrate($results)
38     {
39         $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
40             $this->hydrator,
41             $this->entityPrototype
42         );
43
44         return $hosts->initialize($results->toArray());
45     }
46
47     public function insert($entity)
48     {
49         return parent::insert($this->hydrator->extract($entity));
50     }
51
52     public function updateEntity($entity)
53     {
54         return parent::update(
55             $this->hydrator->extract($entity),
56             $this->idCol . " = " . $entity->getId()
57         );
58     }
```



```
58     }
59
60 }
```

Listing 14.26

Neu sind hier die `insert()` und `updateEntity()`-Methoden. Zusätzlich müssen wir aber auch den `HostHydrator` erweitern, insofern die Spaltenbezeichnungen von den Objekteigenschaften abweichen:

```
1  <?php
2  namespace Helloworld\Mapper;
3
4  use Zend\Stdlib\Hydrator\Reflection;
5  use Helloworld\Entity\Host as HostEntity;
6
7  class HostHydrator extends Reflection
8  {
9      public function hydrate(array $data, $object)
10     {
11         if (!$object instanceof HostEntity) {
12             throw new \InvalidArgumentException(
13                 '$object must be an instance of Helloworld\Entity\Host'
14             );
15         }
16
17         $data = $this->mapField('workstation', 'hostname', $data);
18         return parent::hydrate($data, $object);
19     }
20
21     public function extract($object)
22     {
23         if (!$object instanceof HostEntity) {
24             throw new \InvalidArgumentException(
25                 '$object must be an instance of Helloworld\Entity\Host'
26             );
27         }
28
29         $data = parent::extract($object);
30         $data = $this->mapField('hostname', 'workstation', $data);
31         return $data;
32     }
33 }
```

```
34     protected function mapField($keyFrom, $keyTo, array $array)
35     {
36         $array[$keyTo] = $array[$keyFrom];
37         unset($array[$keyFrom]);
38         return $array;
39     }
40 }
```

Listing 14.27

Im Controller können wir dann einen zuvor geladenen Datensatz ändern:

```
1  <?php
2  $hosts = $this->getServiceLocator()
3      ->get('Helloworld\Mapper\Host')->findByIp('127.0.0.1');
4
5  $host = $hosts->current();
6  $host->setHostname('my-mac');
7
8  $this->getServiceLocator()
9      ->get('Helloworld\Mapper\Host')->updateEntity($host);
```

Listing 14.28

Ähnlich funktioniert so nun auch das Einfügen neuer Datensätze über den Host-Mapper:

```
1  <?php
2  $newEntity = new \Helloworld\Entity\Host();
3  $newEntity->setHostname('michaels-iphone');
4  $newEntity->setIp('192.168.1.56');
5  $this->getServiceLocator()
6      ->get('Helloworld\Mapper\Host')->insert($newEntity);
```

Listing 14.29

Der Vollständigkeit halber hier auch noch einmal die Host-Entity in letzter Ausbaustufe:

```
1  <?php
2  namespace Helloworld\Entity;
3
4  class Host
5  {
6      protected $id;
7      protected $ip;
8      protected $hostname;
9
10     public function getHostname()
11     {
12         return $this->hostname;
13     }
14
15     public function getIp()
16     {
17         return $this->ip;
18     }
19
20     public function setIp($ip)
21     {
22         $this->ip = $ip;
23     }
24
25     public function setHostname($hostname)
26     {
27         $this->hostname = $hostname;
28     }
29
30     public function setId($id)
31     {
32         $this->id = $id;
33     }
34
35     public function getId()
36     {
37         return $this->id;
38     }
39 }
```

Listing 14.30

Gerade die `id`-Eigenschaft ist hier eben noch einmal von Wichtigkeit, weil auf diesem Wege der zu aktualisierende Datensatz bestimmt wird.

14.5 Zend\Db-Alternative: Doctrine 2 ORM

Sind wir mal ganz ehrlich: Zend\Db ist zwar nützlich, stößt aber sehr schnell an seine Grenzen. Die Herausforderung der “Persistenz” in komplexeren Systemen lässt sich damit nicht zufriedenstellend bewältigen. Zend\Db bringt weder eine “Active Record”-Implementierung mit, noch die Funktionalität eines “ORMs”, die aber früher oder später sehr spürbar dazu beiträgt, dass die Komplexität einer Anwendung beherrschbar bleibt. Auch muss man eine Menge Code rund um die “Persistenz” selbst schreiben. Ich empfehle daher unbedingt, einen Blick auf [Doctrine 2](http://www.doctrine-project.org/)⁸ zu werfen. Das Zend Framework 2 und Doctrine 2 ergänzen sich ganz hervorragend und spielen zusammen ihre Stärken erst so richtig aus. Auf der Webseite zum Buch gibt es einen Bereich mit [Zusatzmaterial](http://zendframework2.de/zusatzmaterial.html)⁹. Dort findest du eine Einführungspräsentation von mir zu Doctrine 2, die einen Einstieg in die Materie darstellt.

⁸<http://www.doctrine-project.org/>

⁹<http://zendframework2.de/zusatzmaterial.html>

15 Validatoren

15.1 Standard-Validatoren

Mit `Zend\Validator` bringt das Framework einen einfachen, aber sehr hilfreichen Mechanismus mit, um Werte, etwa Eingaben, die im Rahmen eines POST-Requests durch den Client gesendet wurden, auf definierte Anforderungen hin zu validieren. Für die gängigsten Anforderungen bringt es darüber hinaus auch bereits die konkreten Implementierungen mit. So lässt sich mit wenigen Zeilen Code ein Wert auf Konformität mit dem ISBN-Standard hin überprüfen:

```
1  <?php
2  $validator = new \Zend\Validator\Isbn();
3
4  if($validator->isValid('315000017'))
5      echo "In Ordnung!";
```

Listing 15.1

Wichtig ist die Tatsache, dass es sich hier um eine syntaktische, nicht semantische Überprüfung handelt. Es wird also auch “In Ordnung” ausgegeben, wenn eine ISBN angegeben wird, die zwar korrekt ist, die aber noch keinem Buch gewiesen wurde.

Validatoren bringen üblicherweise Fehlermeldungen für die unterschiedlichen Fehlersituationen mit, die nach einem Validierungsvorgang über `getMessages()` abgerufen werden können:

```
1  <?php
2  $validator = new \Zend\Validator\Isbn();
3
4  if($validator->isValid('315090017'))
5      echo "In Ordnung!";
6  else {
7      foreach ($validator->getMessages() as $messageId => $message) {
8          echo $message;
9      }
10 }
```

Listing 15.2

In diesem Fall sehen wir

```
1 The input is not a valid ISBN number
```

auf dem Schirm. Rufen wir

```
1 <?php
2 $validator = new \Zend\Validator\Isbn();
3
4 if($validator->isValid('12.23'))
5     echo "In Ordnung!";
6 else {
7     foreach ($validator->getMessages() as $messageId => $message) {
8         echo $message;
9     }
10 }
```

Listing 15.3

auf, führt dies hingegen zur Ausgabe

```
1 Invalid type given. String or integer expected
```

Es gibt im Isbn-Validator also diese 2 unterschiedlichen Fehlerfälle. Über die Methode `setMessage()` kann man, wie sich bereits erahnen lässt, individuelle Fehlermeldungen setzen:

```
1 <?php
2 $validator = new \Zend\Validator\Isbn();
3
4 $validator->setMessage(
5     'Es wird ein String oder ein Integer-Wert zur Validierung benötigt!',
6     \Zend\Validator\Isbn::INVALID
7 );
8
9 if($validator->isValid(12.23))
10     echo "In Ordnung!";
11 else {
12     foreach ($validator->getMessages() as $messageId => $message) {
13         echo $message;
14     }
15 }
```

Listing 15.4

Dazu ist jeweils der im Validator verwendete Message-Key anzugeben.

Die folgenden, selbsterklärenden Standard-Validatoren bringt das Framework mit:

- Barcode
- Between
- Callback
- CreditCard
- Crsf
- Date
- DateStep
- Digits
- EmailAddress
- Explode
- GreaterThan
- Hex
- Hostname
- Iban
- Identical
- InArray
- Ip
- Isbn
- LessThan
- NotEmpty
- Regex
- Step
- StringLength
- Uri



Mehrere Validierungen gleichzeitig vornehmen

Mit Hilfe der Klasse `Zend\Validator\ValidatorChain` lassen sich bei Bedarf mehrere Validatoren zur sequentiellen Überprüfung verketteten.

15.2 Eigene Validatoren schreiben

Benötigt man darüber hinaus Validierungsfunktionen, so kann man leicht eigene Validatoren schreiben, wenn diese von `AbstractValidator` erben:

```
1  <?php
2  namespace Helloworld\Validator;
3
4  use Zend\Validator\AbstractValidator;
5
6  class Float extends AbstractValidator
7  {
8      const FLOAT = 'float';
9
10     protected $messageTemplates = array(
11         self::FLOAT => "'%value%' is not a float value."
12     );
13
14     public function isValid($value)
15     {
16         $this->setValue($value);
17
18         if (!is_float($value)) {
19             $this->error(self::FLOAT);
20             return false;
21         }
22
23         return true;
24     }
25 }
```

Listing 15.5

16 Webforms

Webforms sind integraler Bestandteil von Webanwendungen, sind sie im Grunde doch die einzige Möglichkeit, wie ein Anwender Daten von sich zum Server, sprich zur Anwendung, übertragen kann. Für den Anwendungsentwickler bringen Webforms immer vor allem vielfältige Herausforderungen mit sich. Ein gutes Web-Framework hat umfassende Unterstützung für Webforms an Board und treibt ihnen so ein wenig den Schrecken aus. Das Zend Framework 2 bietet mit `Zend\Form` eine leistungsstarke Lösung, die so ziemlich alles kann, was man sich so wünscht. Auch, wenn sie nicht ganz so leicht zu verstehen ist.

Vorab ein wenig Theorie: Jede Webform basiert grundsätzlich auf einem oder mehreren Objekten vom Typ `Zend\Form\Element`. Sie bilden die atomare Einheit von `Zend\Form`-basierten Formularen. Ein Element entspricht dabei zunächst grundsätzlich erst einmal den üblichen HTML-Formularelementen, die man so kennt, also Eingabefelder (`text`), Radiobuttons (`radio`), Auswahllisten (`select`) und so weiter. Für jedes Element kann über einen sog. "Input" festgelegt werden, welche Filter auf die jeweils eingegebenen Daten angewendet werden sollen, sobald diese empfangen wurden und welche Regeln für die Validierung der Daten herangezogen werden sollen. Typische Filter sind etwa `StripTags`, der sämtliche HTML-Tags in den eingegebenen Zeichenketten entfernt oder `StringToLower`, der - man vermutet es schon - alle eingegebenen Zeichen in Kleinbuchstaben konvertiert. Validierung meint die Überprüfung der eingegebenen Daten auf zuvor definierte Bedingungen. So lässt sich für ein Eingabefeld etwa der `Isbn-Validator` nutzen, der überprüft, ob die Daten einer ISBN-Nummer entsprechen, während `NotEmpty` darauf prüft, ob zumindest irgendwas in das Feld eingetragen wurde. Die einzelnen "Inputs" werden im sog. "InputFilter" aggregiert und der jeweiligen Webform zur Verfügung gestellt. Die einzelnen Form-Elemente können wiederum semantisch zu sogenannten "Fieldsets" gruppiert werden. Ein `Zend\Form\Fieldset` ist dabei übrigens technisch wieder auch nur ein "Element", leitet es doch von der Klasse `Zend\Form\Element` ab. "Fieldsets" oder einzelne Elemente (oder auch beides zusammen) werden dann wiederum zu einer `Zend\Form\Form` kombiniert, die technisch wieder auch nur ein "Element" ist - auch sie leitet von `Zend\Form\Element` ab.

16.1 Eine Form erstellen

Eine Webform mit HTML ist schnell erstellt:

```
1 <form action="#" method="post">
2     <fieldset>
3         <label for="name">Ihr Name:</label>
4         <input type="text" id="name" />
5         <label for="email">Ihre E-Mail-Adresse:</label>
6         <input type="email" id="email" />
7         <input type="submit" value="Eintragen" />
8     </fieldset>
9 </form>
```

Listing 16.1

Doch der erfahrene Anwendungsentwickler weiß, damit ist es noch lange nicht getan: Die empfangenen Daten wollen syntaktisch und semantisch validiert, für die weitere Verarbeitung - etwa die Speicherung in einer Datenbank - aufbereitet und Fehlersituationen mit probaten Mitteln behandelt werden, nicht zuletzt weil clientseitige Überprüfungen der eingegebenen Daten, allein schon aus Sicherheitsgründen, niemals ausreichend sind. All diesen Code müsste man normalerweise von Hand entwickeln, den entsprechenden Code etwa im Controller realisieren. Hier kann das Framework helfen. Die grundsätzliche Idee ist es, eine Webform als eigenständiges Konstrukt zu realisieren, dass alle Informationen über die enthaltenen Felder und dessen syntaktischen und semantischen Rahmenbedingungen enthält, sowie über das notwendige Wissen darüber verfügt, wie Daten hinein und auf dem besten Wege auch wieder hinauskommen. Im Framework kommen eine Reihe verschiedener Klassen und Komponenten für Webforms zum Einsatz:

- `Zend\Form`: Kernkomponente, über die der Anwendungsentwickler mit Webforms interagiert.
- `Zend\InputFilter`: Erweitert Webforms um Filter- und Validierungsmöglichkeiten.
- `Zend\View\Helper`: Ermöglicht die visuelle Darstellung von Formularen.
- `Zend\Stdlib\Hydrator`: Erweitert Webforms um die Möglichkeit, Daten aus der Form automatisch in andere Objekte zu überführen oder von dort zu importieren.

Das Framework überlässt dem Anwendungsentwickler einige Entscheidungen darüber, wie eine Webform "zusammengesteckt" wird. Dabei schwingt das Pendel - wie so häufig im Zend Framework 2 - zwischen dem Schreiben von Code und dem Schreiben von Konfiguration. Abgesehen davon gibt es eine ganze Reihe an Möglichkeiten, um mit Webforms umzugehen. Eine empfehlenswerte Herangehensweise ist es, Formulare über eigene Form-Klassen abzubilden. Wenn wir die oben gezeigte Form für die Anmeldung zu einem Newsletter über ein `Zend\Form`-Objekt abbilden wollen würden, müssen wir in der Datei `src/Helloworld/Form/SignUp.php` die Webform definieren:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Form;
5
6  class SignUp extends Form
7  {
8      public function __construct()
9      {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13
14         $this->add(array(
15             'name' => 'name',
16             'attributes' => array(
17                 'type' => 'text',
18                 'id' => 'name'
19             ),
20             'options' => array(
21                 'label' => 'Ihr Name:'
22             ),
23         ));
24
25         $this->add(array(
26             'name' => 'email',
27             'attributes' => array(
28                 'type' => 'email',
29                 'id' => 'email'
30             ),
31             'options' => array(
32                 'label' => 'Ihre E-Mail-Adresse:'
33             ),
34         ));
35
36         $this->add(array(
37             'name' => 'submit',
38             'attributes' => array(
39                 'type' => 'submit',
40                 'value' => 'Eintragen'
41             ),
42         ));
```

```
43         }  
44     }
```

Listing 16.2

Über die `add()`-Methode werden der Form neue Elemente oder Fieldsets hinzugefügt. Dazu ruft die `add()`-Methode wiederum die `Zend\Form\Factory` auf, die es versteht, die in Form eines Arrays übergebene Spezifikation („Spec“ genannt) zu interpretieren und das entsprechende Element zu erzeugen. Die wichtigsten Bestandteile der Spezifikation sind `name`, `type`, `attributes` und `options`. Während der `name` frei wählbar ist, repräsentiert der `type` einen tatsächlich im Framework existierenden Element-Typen (in diesem Fall werden Elemente vom Typ `Zend\Form\Element\Text` erzeugt, weil nichts anders angegeben ist) und `attributes` die Aufstellung aller Attribute des am Schluss erzeugten HTML-Elements (`<input name="name" type="text" id="name">`). Der Abschnitt `options` erlaubt die Definition des Feld-Labels (also das, was zusätzlich erklärend vor oder über dem eigentlichen Eingabefeld angezeigt wird) via `label` bzw. wiederum auch die Attribute des Labels via `label_attributes`.

Zusätzlich zu den einzelnen Elementen werden über `setAttribute()` einige Attribute für das Formular selbst festgelegt, darunter `action` und `method`, die beide unabdingbar dafür sind, dass die Webform später auch abgeschickt werden kann.

16.2 Eine Form anzeigen

Für alle, die die Version 1 des Frameworks eingesetzt haben, habe ich eine gute Nachricht: Es gibt keine Form-Dekoratoren mehr. Form-Dekoratoren waren ein sehr engagierter aber gleichzeitig auch hochkomplexer Ansatz, das Rendering von Webforms über die Schachtelung von View-Objekten zu ermöglichen, die jeweils einen kleinen Teil des finalen HTML-Markups erzeugt haben. Ich würde behaupten, dass Form-Dekoratoren so ziemlich das Schwierigste waren, was das Zend Framework 1 in sich hatte und auch sonst eine Vielzahl von Problemen generiert haben, an die man vorher nicht mal gedacht hatte. Aber genug der Vergangenheit; in Version 2 sind sie nicht mehr dabei. Der Vorteil ist ganz klar die einfachere Nutzung der Formulare, allerdings nur mit Bordmitteln so ein wenig auf Kosten von kompaktem Code. Um das oben definierte Formular anzuzeigen, ist nämlich jetzt etwas mehr View-Code notwendig, als das `<?php echo $this->form; ?>` vergangener Tage (allerdings lässt sich auch das mit wenig Mühe realisieren, wie wir im Praxisteil dieses Buches noch sehen werden):

```
1 <?php
2 $this->form->prepare();
3 echo $this->form()->openTag($this->form);
4 echo $this->formRow($this->form->get('name'));
5 echo $this->formRow($this->form->get('email'));
6 echo $this->formSubmit($this->form->get('submit'));
7 echo $this->form()->closeTag();
```

Listing 16.3

Dieser Code muss in das View-File der Action, in der die Form instanziiert und im Rahmen des ViewModel zurückgeliefert wird:

```
1 <?php
2 return new ViewModel(
3     array(
4         'form' => new \Helloworld\Form\SignUp()
5     )
6 );
```

Listing 16.4

Für die Darstellung kommen eine Reihe von “View Helfern” und die Form selbst zum Einsatz. Grob gesagt gibt es für jeden Typ von Formularelement, den HTML definiert, den entsprechenden ViewHelper, der für die Darstellung verwendet werden kann. Zudem gibt es einige weitere Hilfskonstrukte, wie FormRow, das dafür sorgt, dass ein Formularfeld mit seinem “Label”, dem eigentlichen Feld und ggf. einer vorhandenen Fehlermeldung sinngemäß “in einer Reihe” angezeigt wird. Die View Helper sind allesamt so gut wie selbsterklärend. Wichtig ist, dass prepare() zu Beginn aufgerufen wird, noch bevor auf andere Elemente zugegriffen wird. Ansonsten führt dies zu einem Fehler.

16.3 Formulareingaben verarbeiten

Im Grunde gibt es zwei Möglichkeiten, die Formularverarbeitung zu realisieren: Entweder in der gleichen Action, in der auch initial die leere Webform erzeugt wurde, oder aber in einer separaten Action. Soll die Verarbeitung in der gleichen Action stattfinden, lässt sich durch eine “isPost()-Überprüfung” feststellen, ob das Formular entweder angezeigt oder Eingaben verarbeitet werden sollen:

```
1  <?php
2  if ($this->getRequest()->isPost()) {
3      // Formularverarbeitung
4  } else {
5      return new ViewModel(
6          array(
7              'form' => new \Helloworld\Form\SignUp()
8          )
9      );
10 }
```

Listing 16.5

Um auf die abgesendeten Daten zuzugreifen, kann man nun entweder direkt auf die POST-Daten zugreifen:

```
1  <?php
2  $form = new \Helloworld\Form\SignUp();
3
4  if ($this->getRequest()->isPost()) {
5      $data = $this->getRequest()->getPost();
6      var_dump($data);exit;
7  } else {
8      return new ViewModel(
9          array(
10             'form' => $form
11          )
12      );
13 }
```

Listing 16.6

Folgende Ausgabe wird erzeugt:

```
1  class Zend\Stdlib\Parameters#74 (3) {
2      public $name => string(13) "Michael Romer"
3      public $email => string(24) "zf2buch@michael-romer.de"
4      public $submit => string(9) "Eintragen"
5  }
```

Oder aber man kann über das Form-Objekt auf die gesendeten Daten zugreifen. Das geht allerdings nur dann, wenn man zuvor die Daten über die Webform hat validieren lassen.

16.4 Formulareingaben validieren

Bis hier hin ist der Vorteil von `Zend\Form` zugegebenermaßen relativ überschaubar. Aber jetzt geht es los: Wenn wir die abgesendeten Daten nicht über das POST-Array von PHP beziehen, sondern über die Form selbst, können die Eingaben auf einfache Weise auf Basis von zuvor definierter Regeln automatisch validiert und Daten gefiltert werden.

Dazu muss zunächst der sog. “InputFilter” definiert werden. Die Bezeichnung “InputFilter” ist ein wenig irreführend, werden über ihn doch nicht nur Filter, sondern auch die Validatoren definiert. Während Filter die Eingabedaten bei Bedarf modifizieren, testen Validatoren die Daten auf bestimmte Gegebenheiten, etwa eine maximale Zeichenlänge.

Für die Definition des “InputFilter” gibt es mehrere Möglichkeiten. So können die Definitionen etwa in eine eigene Klasse wandern, oder aber sie werden über eine Methode `getInputFilter()` von der Form-Klasse selbst bereitgestellt. In diesem Fall realisieren wir den “InputFilter” in einer eigenen Klasse in der Datei `src/Helloworld/Form/SignUpFilter.php`:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Form;
5  use Zend\InputFilter\InputFilter;
6
7  class SignUpFilter extends InputFilter
8  {
9      public function __construct()
10     {
11         $this->add(array(
12             'name' => 'email',
13             'required' => true,
14             'validators' => array(
15                 array(
16                     'name' => 'EmailAddress'
17                 )
18             ),
19         ));
20
21         $this->add(array(
22             'name' => 'name',
23             'required' => true,
24             'filters' => array(
25                 array(
26                     'name' => 'StringTrim'
```

```

27         )
28     )
29     ));
30 }
31 }

```

Listing 16.7

Über die `add()`-Methode werden hier die einzelnen “Inputs” zum “InputFilter” hinzugefügt. Ein “Input” entspricht dabei im Grunde genommen einem Formularelement, also einer “Eingabemöglichkeit”, das über den `name` referenziert und später dann berücksichtigt wird. Über `required` lässt sich steuern, ob es sich um ein Pflichtfeld handelt. Über `validators` und `filters` lassen sich entsprechend die anzuwendenden Filter und Validatoren definieren, wobei dessen `name` mit den vom Framework mitgelieferten Filtern oder Validatoren bzw. selbst nachträglich hinzugefügten Filtern oder Validatoren korrespondieren muss. Ein Blick in den Quellcode des `Zend\Validator\ValidatorPluginManager` offenbart die standardmäßig verfügbaren Validatoren und deren symbolische Namen, über die sie anzusprechen sind. `Zend\Filter\FilterPluginManager` gibt Auskunft über die Filter des Frameworks.

Jetzt müssen wir aber erst einmal noch die Form-Definition um den `SignUpFilter` erweitern:

```

1  <?php
2  namespace HelloWorld\Form;
3
4  use Zend\Form\Form;
5
6  class SignUp extends Form
7  {
8      public function __construct()
9      {
10         parent::__construct('signup');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13         $this->setInputFilter(new \HelloWorld\Form\SignUpFilter());
14
15         $this->add(array(
16             'name' => 'name',
17             'attributes' => array(
18                 'type' => 'text',
19             ),
20             'options' => array(
21                 'id' => 'name',
22                 'label' => 'Ihr Name:'

```



```

23         ),
24     ));
25
26     $this->add(array(
27         'name' => 'email',
28         'attributes' => array(
29             'type' => 'email',
30         ),
31         'options' => array(
32             'id' => 'email',
33             'label' => 'Ihre E-Mail-Adresse:'
34         ),
35     ));
36
37     $this->add(array(
38         'name' => 'submit',
39         'attributes' => array(
40             'type' => 'submit',
41             'value' => 'Eintragen'
42         ),
43     ));
44 }
45 }

```

Listing 16.8

Die Methode `setInputFilter()` verknüpft hier die Form mit dem “InputFilter”. Auch hier gilt natürlich eigentlich der allgemeine Rat, den `SignUpFilter` besser nicht hart zu verdrahten, sondern über eine entsprechende Factory und den `ServiceManager` oder aber `Zend\Di` zu injizieren.

Nun lässt sich im Controller die Formverarbeitung wie folgt bewerkstelligen:

```

1  <?php
2  public function indexAction()
3  {
4      $form = new \Helloworld\Form\SignUp();
5
6      if ($this->getRequest()->isPost()) {
7          $form->setData($this->getRequest()->getPost());
8
9          if ($form->isValid()) {
10             var_dump($form->getData());
11         } else {

```

```
12         return new ViewModel(  
13             array(  
14                 'form' => $form  
15             )  
16         );  
17     }  
18     } else {  
19         return new ViewModel(  
20             array(  
21                 'form' => $form  
22             )  
23         );  
24     }  
25 }
```

Listing 16.9

Eine erfolgreiche Eingabe führt in diesem Fall zur erfreulichen Ausgabe:

```
1 array(2) {  
2     ["email"]=> string(24) "zf2buch@michael-romer.de"  
3     ["name"]=> string(13) "Michael Romer"  
4 }
```

Die Daten liegen nun also zur weiteren Verarbeitung in Form eines Arrays vor. Etwaige Filter, die für die jeweiligen Felder registriert wurden, haben an diese Stelle bereits ihre Anwendung gefunden.

Auf diese Art und Weise werden nun übrigens auch bereits Fehlermeldungen angezeigt, denn wird das Formular nicht validiert, geben wir es zur erneuten Anzeige an die View. Sendet man das Formular einmal ohne Eingaben ab (was auf Basis der SignUpFilter-Definition zu einem Fehler führen sollte), sieht man bei den beiden Feldern entsprechend:

```
1 "Value is required and can't be empty"
```

Um die angezeigte Fehlermeldung zu verändern, kommt das options-Array bei der Validator-Definition zum Einsatz. Bei der oben gezeigten Fehlermeldung gibt es allerdings direkt eine Besonderheit, denn der SignUpFilter verwendet aktuell noch gar nicht den NotEmpty-Validator, den wir mit den notwendigen "options" ausstatten könnten. Sondern die Verpflichtung, das Feld ausfüllen zu müssen, wird über 'required' => true zum Ausdruck gebracht. Das Framework erzeugt auf Grund dessen automatisch den entsprechenden Validator. Der Ausdruck 'required' => true versteht sich also als "Shortcut". Wenn wir den Filter wie folgt anpassen, können wir individuelle Fehlermeldungen hinterlegen:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Form;
5  use Zend\InputFilter\InputFilter;
6
7  class SignUpFilter extends InputFilter
8  {
9      public function __construct()
10     {
11         $this->add(array(
12             'name' => 'email',
13             'validators' => array(
14                 array(
15                     'name' => 'NotEmpty',
16                     'options' => array(
17                         'messages' => array(
18                             \Zend\Validator\NotEmpty::IS_EMPTY =>
19                             'Bitte geben Sie etwas ein.'
20                         )
21                     )
22                 ),
23                 array(
24                     'name' => 'EmailAddress',
25                     'options' => array(
26                         'messages' => array(
27                             \Zend\Validator\EmailAddress::INVALID_FORMAT =>
28                             'Bitte richtige E-Mail-Adresse eingeben.'
29                         )
30                     )
31                 )
32             ),
33         ));
34
35         $this->add(array(
36             'name' => 'name',
37             'filters' => array(
38                 array(
39                     'name' => 'StringTrim'
40                 )
41             ),
42             'validators' => array(
```

```

43         array(
44             'name' => 'NotEmpty',
45             'options' => array(
46                 'messages' => array(
47                     \Zend\Validator\NotEmpty::IS_EMPTY =>
48                     'Bitte geben Sie etwas ein.'
49                 )
50             )
51         )
52     );
53 });
54 }
55 }

```

Listing 16.10

Senden wir nun das Formular ab, ohne eine E-Mail-Adresse einzugeben, so bekommen wir die folgenden Fehlermeldung:

- 1 Bitte geben Sie etwas ein.
- 2 Bitte richtige E-Mail-Adresse eingeben.

Oder besser gesagt: Wir sehen gleich 2 Fehlermeldungen. Man muss wissen, dass in der Verarbeitung Validator für Validator abgearbeitet wird und etwaige Fehlermeldungen gesammelt werden. Es kommen dann auch alle Fehlermeldungen zur Anzeige, wenn der entsprechende “View Helper” zum Einsatz kommen. Meistens ist das aber nicht wirklich hilfreich für den Anwender, sondern eine einzige Fehlermeldung ausreichend. Dafür kann die Option 'break_chain_on_failure' => true gesetzt werden. Sie sorgt dafür, dass nach einem fehlgeschlagenen Validator die folgenden erst gar nicht mehr ausgeführt und keine etwaigen zusätzlichen Fehlermeldungen erzeugt werden:

```

1  <?php
2  namespace HelloWorld\Form;
3
4  use Zend\Form\Form;
5  use Zend\InputFilter\InputFilter;
6
7  class SignUpFilter extends InputFilter
8  {
9      public function __construct()
10     {
11         $this->add(array(
12             'name' => 'email',

```

```
13         'validators' => array(
14             array(
15                 'name' => 'NotEmpty',
16                 'break_chain_on_failure' => true,
17                 'options' => array(
18                     'messages' => array(
19                         \Zend\Validator\NotEmpty::IS_EMPTY =>
20                         'Bitte geben Sie etwas ein.'
21                     )
22                 )
23             ),
24             array(
25                 'name' => 'EmailAddress',
26                 'options' => array(
27                     'messages' => array(
28                         \Zend\Validator\EmailAddress::INVALID_FORMAT =>
29                         'Bitte richtige E-Mail-Adresse eingeben.'
30                     )
31                 )
32             ),
33         ),
34     ));
35
36     $this->add(array(
37         'name' => 'name',
38         'filters' => array(
39             array(
40                 'name' => 'StringTrim'
41             )
42         ),
43         'validators' => array(
44             array(
45                 'name' => 'NotEmpty',
46                 'options' => array(
47                     'messages' => array(
48                         \Zend\Validator\NotEmpty::IS_EMPTY =>
49                         'Bitte geben Sie etwas ein.'
50                     )
51                 )
52             )
53         )
54     ));
```

```
55         }  
56     }
```

Listing 16.11

Da einzelne Validatoren je nach Situation unterschiedliche Fehlermeldungen generieren können, muss die “message” zum passenden Key hinterlegt werden, der über eine Konstante in der jeweiligen Klasse zugänglich ist. Im Zweifelsfall hilft auch der Blick in den Code des jeweiligen Validators.

16.5 Standard-Formelemente

So wie in den Beispielen zuvor gezeigt, kann man problemlos beliebige HTML-Elemente mit den Filtern und Validatoren der Wahl erzeugen. Allerdings kann dieses Vorgehen je nach Element eine Menge Arbeit machen, eben weil alle Konfigurationen manuell vorgenommen werden müssen. Hier hat das Framework noch “etwas in petto”: Eine ganze Reihe “vorkonfektionierter” Elemente, die gleich die notwendige Konfiguration mitbringen:

- Button
- Captcha
- Checkbox
- Collection
- Color
- Csrf
- Date
- DateTime
- DateTimeLocal
- Email
- File
- Hidden
- Image
- Month
- MultiCheckbox
- Number
- Password
- Radio
- Range
- Select
- Submit
- Text
- Textarea

- Time
- URL
- Week

Nehmen wir als Beispiel das Element Number. Wir wollen ein Eingabefeld erzeugen, in das numerische Werte eines definierten Bereichs eingetragen werden dürfen. Wenn wir diese Regeln selbst aufbauen wollen würden, so müssten wir eine ganze Reihe von Validatoren konfigurieren, darunter:

- NumberValidator: Es sollen nur Zahlen eingegeben werden können.
- GreaterThanValidator: Der eingegebene Wert muss oberhalb oder gleich des Minimums sein.
- LessThanValidator: Der eingegebene Wert muss unterhalb oder gleich des Maximums sein.
- StepValidator: Es sollen nur ganzzahlige Werte angenommen werden.

Diese Arbeit können wir uns sparen, wenn wir direkt das `Zend\Form\Element\Number` einsetzen:

```
1  <?php
2  // [...]
3  $this->add(array(
4      'name' => 'age',
5      'type' => 'Zend\Form\Element\Number',
6      'attributes' => array(
7          'id' => 'age',
8          'min' => 18,
9          'max' => 99,
10         'step' => 1
11     ),
12     'options' => array(
13         'label' => 'Wieviel Jahre sind sie alt?'
14     ),
15 ));
16 // [...]
```

Listing 16.12

Je nach Browser und dessen HTML5-Unterstützung fällt auch gleich auf, dass auf Basis der Regeln nicht nur die serverseitige Überprüfung gut funktioniert, sondern fehlerhafte Eingaben auch direkt im Client moniert werden. Die Konfiguration im `attributes`-Array wird nämlich nicht nur vom entsprechenden “View Helper” später bei der Erzeugung des HTML-Codes berücksichtigt, sondern auch von den Validatoren verwendet.

Übrigens hängt das Number-Element auch gleich einen `Zend\Filter\StringTrim` an, so dass auch das nicht mehr im eigenen Code geschehen muss.

16.6 Fieldsets

Dem wachsamem Auge ist vielleicht bereits aufgefallen, dass wir die ursprüngliche Webform, die wir händisch erzeugt haben, mit unserer “Objekt-Version” nicht zu 100% nachgebaut haben. Das “Fieldset” fehlt. Eine Sache an dieser Stelle vorweg: Wenn ich im Folgenden von “Fieldsets” spreche, meine ich explizit nicht das zuvor genannte, fehlende HTML-Fieldset, sondern “Fieldsets” in der Definition des Zend Frameworks, die grundsätzlich erst einmal nichts mit HTML-Fieldsets zu tun haben müssen, bei Bedarf aber können. Diese Unterscheidung ist wirklich wichtig für das Verständnis.

Ein “Fieldset” dient der Gruppierung einzelner Felder. Das ist insbesondere dann sinnvoll, wenn sich eine Form aus Elementen unterschiedlicher Belange zusammensetzt. Wenn wir etwa das Formular von oben um Eingabefelder für die Anschrift des Nutzers erweitern würden, so wären diese Daten vermutlich in einer eigenen Entity verwaltet, einer eigenen Datenbanktabelle gespeichert und von der eigentlichen User-Entity bzw. Tabelle referenziert. Der handwerkliche Vorteil von Fieldsets ist die Tatsache, dass diese in unterschiedlichen Formularen wiederverwendet werden können. Wenn wir bei dem Beispiel bleiben, so bedeutet dies, dass der Nutzer seine Anschrift im Rahmen der “Newsletter-Anmeldung” angeben würde (tun wir einfach mal so, also wäre das sinnvoll), aber auch später im Kundenkonto über das “Anschrift-Ändern-Formular” anpassen könnte. Beide Male käme dabei das einmalig definierte “Fieldset” zum Einsatz. Schauen wir uns exemplarisch eine “Fieldset”-Definition für die beiden Felder des Formulars an:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Fieldset;
5
6  class UserFieldset extends Fieldset
7  {
8      public function __construct()
9      {
10         parent::__construct('user');
11
12         $this->add(array(
13             'name' => 'name',
14             'attributes' => array(
15                 'type' => 'text',
16                 'id' => 'name'
17             ),
18             'options' => array(
19                 'id' => 'name',
20                 'label' => 'Ihr Name:',
21             )
22         ));
23     }
24 }
```



```

22         ));
23
24         $this->add(array(
25             'name' => 'email',
26             'attributes' => array(
27                 'type' => 'email',
28             ),
29             'options' => array(
30                 'id' => 'email',
31                 'label' => 'Ihre E-Mail-Adresse:'
32             ),
33         ));
34     }
35 }

```

Listing 16.13

Die Definition des “Fieldset” hat große Ähnlichkeit mit der des Formulars von weiter oben, außer allerdings der Tatsache, dass sich das “Fieldset” nach Außen hin alleine nicht mehr als Webform präsentiert, sondern von einer `Zend\Form` inkludiert werden muss, um es zur Anzeige zu bringen:

```

1  <?php
2  namespace HelloWorld\Form;
3
4  use Zend\Form\Form;
5
6  class SignUp extends Form
7  {
8      public function __construct()
9      {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13         $this->setInputFilter(new \HelloWorld\Form\SignUpFilter());
14
15         $this->add(new \HelloWorld\Form\UserFieldset());
16
17         $this->add(array(
18             'name' => 'submit',
19             'attributes' => array(
20                 'type' => 'submit',
21                 'value' => 'Eintragen'
22             ),

```

```
23         ));  
24     }  
25 }
```

Listing 16.14

Hier also wieder die SignUp-Form, allerdings diesmal mit dem `UserFieldset`, statt dessen einzelner Felder, die sich nun ja gruppiert im `UserFieldset` befinden. Damit die Form weiterhin korrekt angezeigt wird, müssen wir noch den View-Code anpassen:

```
1  <?php  
2  $this->form->prepare();  
3  echo $this->form()->openTag($this->form);  
4  echo $this->formRow($this->form->get('user')->get('name'));  
5  echo $this->formRow($this->form->get('user')->get('email'));  
6  echo $this->formSubmit($this->form->get('submit'));  
7  echo $this->form()->closeTag();
```

Listing 16.15

Wir greifen jetzt über `get('user')` zunächst auf das Fieldset zu und von dort aus dann auf die Elemente des Fieldsets. Nehmen wir diese Anpassung hier nicht vor, so bekommen wir einen Fehler. Soweit, so gut. Aber noch sind wir nicht ganz fertig, denn unsere Form validiert nun nicht mehr korrekt, weil der `SignUpFilter` noch der Webform zugewiesen ist, dort die Konfigurationen aber bereits nicht mehr zutreffen:

```
1  <?php  
2  // [...]   
3  $this->setInputFilter(new \Helloworld\Form\SignUpFilter());  
4  // [...]
```

Listing 16.16

Wir müssen nun also dafür sorgen, dass das `UserFieldset` selbst über die notwendigen Konfigurationen verfügt. Das machen wir, indem wir dem `UserFieldset` die Methode `getInputFilterSpecification()` spendieren und dort die notwendigen Konfigurationen vornehmen:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Fieldset;
5
6  class UserFieldset extends Fieldset
7  {
8      public function __construct()
9      {
10         parent::__construct('user');
11
12         $this->add(array(
13             'name' => 'name',
14             'attributes' => array(
15                 'type' => 'text',
16                 'id' => 'name'
17             ),
18             'options' => array(
19                 'id' => 'name',
20                 'label' => 'Ihr Name:',
21             )
22         ));
23
24         $this->add(array(
25             'name' => 'email',
26             'attributes' => array(
27                 'type' => 'email',
28             ),
29             'options' => array(
30                 'id' => 'email',
31                 'label' => 'Ihre E-Mail-Adresse:'
32             ),
33         ));
34     }
35
36     public function getInputFilterSpecification()
37     {
38         return array(
39             'email' => array(
40                 'validators' => array(
41                     array(
42                         'name' => 'NotEmpty',
```

```

43         'break_chain_on_failure' => true,
44         'options' => array(
45             'messages' => array(
46                 \Zend\Validator\NotEmpty::IS_EMPTY =>
47                 'Bitte geben Sie etwas ein.'
48             )
49         )
50     ),
51     array(
52         'name' => 'EmailAddress',
53         'options' => array(
54             'messages' => array(
55                 \Zend\Validator\EmailAddress::INVALID_FORMAT
56                 => 'Bitte richtige E-Mail-Adresse eingeben.'
57             )
58         )
59     ),
60 ),
61 ),
62 'name' => array(
63     'filters' => array(
64         array(
65             'name' => 'StringTrim'
66         )
67     ),
68     'validators' => array(
69         array(
70             'name' => 'NotEmpty',
71             'options' => array(
72                 'messages' => array(
73                     \Zend\Validator\NotEmpty::IS_EMPTY =>
74                     'Bitte geben Sie etwas ein.'
75                 )
76             )
77         )
78     )
79 );
80 };
81 }
82 }

```

Listing 16.17

So. Was haben wir jetzt hier gemacht? Wir haben die Definitionen aus dem `SignUpFilter` entnommen und direkt in die Methode `getInputFilterSpecification()` des “Fieldsets” übertragen. Dies ist notwendig, weil die dem “Fieldset” übergeordnete Form jeweils alle “InputFilter”-Spezifikationen über die `getInputFilterSpecification()`-Methoden der referenzierten “Fieldsets” bezieht.

Nun erhalten wir das folgende, gewünschte Ergebnis, wenn das Formular mit gültigen Eingaben abgeschickt wird:

```
1 array(2) {  
2     'submit' => string(9) "Eintragen"  
3     'user' => array(2) {  
4         'name' => string(13) "Michael Romer"  
5         'email' => string(24) "zf2buch@michael-romer.de"  
6     }  
7 }
```

16.7 Entities mit Forms verbinden

In aller Regel stehen Webforms einer Anwendung in direkter Beziehung mit dessen Entities. Ein “Product” wird samt Mengenangabe in den Warenkorb gelegt und so eine “Order” erzeugt, ein “User” bei der Anmeldung im System registriert, die “ShippingAddress” im Rahmen des Checkout erfasst und so weiter. Deshalb ist man häufig damit beschäftigt, die validierten Daten aus einer Webform in die passende Entity zu überführen, die dann etwa in der Datenbank persistiert wird. Oder anders herum werden aus der Datenbank geladene Daten einer Entity in eine Webform übertragen, etwa um dessen Eigenschaften dort editierbar zu machen.

Um diesen Vorgang zu vereinfachen, kommen die sog. “Hydrators” zum Einsatz, die wir bereits im Rahmen von `Zend\Db` kennengelernt haben. Hier schließt sich also der Kreis, wenn man so will. Zunächst benötigen wir für die Form die passende User-Entity:

```
1 <?php  
2 namespace Helloworld\Entity;  
3  
4 class User  
5 {  
6     protected $id;  
7     protected $email;  
8     protected $name;  
9  
10    public function setEmail($email)  
11    {  
12        $this->email = $email;
```

```
13     }
14
15     public function getEmail()
16     {
17         return $this->email;
18     }
19
20     public function setId($id)
21     {
22         $this->id = $id;
23     }
24
25     public function getId()
26     {
27         return $this->id;
28     }
29
30     public function setName($name)
31     {
32         $this->name = $name;
33     }
34
35     public function getName()
36     {
37         return $this->name;
38     }
39 }
```

Listing 16.18

Damit diese Entity als “Datencontainer” von der SignUp-Form verwendet wird, muss man an entsprechender Stelle die dafür notwendige Konfiguration vornehmen. In diesem Beispiel arbeiten wir jetzt zunächst wieder ohne “Fieldsets”. Wir haben es also mit einem “normalen” Formular zu tun, in dem sich direkt die einzelnen Elemente befinden:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Form;
5
6  class SignUp extends Form
7  {
8      public function __construct()
9      {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13
14         $this->add(array(
15             'name' => 'name',
16             'attributes' => array(
17                 'type' => 'text',
18                 'id' => 'name'
19             ),
20             'options' => array(
21                 'id' => 'name',
22                 'label' => 'Ihr Name:',
23             )
24         ));
25
26         $this->add(array(
27             'name' => 'email',
28             'attributes' => array(
29                 'type' => 'email',
30             ),
31             'options' => array(
32                 'id' => 'email',
33                 'label' => 'Ihre E-Mail-Adresse:'
34             ),
35         ));
36
37         $this->add(array(
38             'name' => 'submit',
39             'attributes' => array(
40                 'type' => 'submit',
41                 'value' => 'Eintragen'
42             ),
```

```
43         ));
44     }
45 }
```

Listing 16.19

Die Definitionen von Validatoren und Filtern bleiben der Übersichtlichkeit halber hier Außen vor. Im Controller binden wir nun eine Entity und konfigurieren den zu verwendenden Hydrator:

```
1  <?php
2
3  namespace HelloWorld\Controller;
4
5  use Zend\Mvc\Controller\AbstractActionController;
6  use Zend\View\Model\ViewModel;
7
8  class IndexController extends AbstractActionController
9  {
10     public function indexAction()
11     {
12         $form = new \HelloWorld\Form\SignUp();
13         $form->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
14         $form->bind(new \HelloWorld\Entity\User());
15
16         if ($this->getRequest()->isPost()) {
17             $form->setData($this->getRequest()->getPost());
18
19             if ($form->isValid()) {
20                 var_dump($form->getData());
21             } else {
22                 return new ViewModel(
23                     array(
24                         'form' => $form
25                     )
26                 );
27             }
28         } else {
29             return new ViewModel(
30                 array(
31                     'form' => $form
32                 )
33             );
34         }
35     }
36 }
```



```
35     }  
36 }
```

Listing 16.20

Schicken wir nun die Webform mit validen Daten ab, so erhalten wir via `$form->getData()` eine fertig gefüllte Entity zurück (statt vorher ein Array):

```
1 class Helloworld\Entity\User#186 (3) {  
2     protected $id => NULL  
3     protected $email => string(24) "zf2buch@michael-romer.de"  
4     protected $name => string(13) "Michael Romer"  
5 }
```

Die maßgebliche Konfiguration ist der Aufruf von `bind()`, bei dem wir der Form das Objekt (bzw. dessen Klasse) übergeben, in das die Daten überführt werden sollen, dass in diesem Fall mit Hilfe des zuvor definierten Reflection-Hydrators passiert. Die folgenden “Hydrator” bringt das Framework standardmäßig mit:

- **ArraySerializable:** Dies ist der Standard-Hydrator, den `Zend\Form` verwendet, wenn nichts anderes definiert ist. Er setzt voraus, dass das jeweilige Objekt die Methoden `getArrayCopy()` und `exchangeArray()` bzw. `populate()` implementiert und auf diesem Wege die notwendigen Informationen zur Verfügung stellt.
- **ClassMethods:** Verwendet die Getter/Setter-Methoden des Objektes (bzw. der Klasse), um die entsprechenden Daten einzufügen (`hydrate()`) bzw. auszulesen (`extract()`).
- **ObjectProperty:** Bedient sich der öffentlichen Eigenschaften des Objektes.
- **Reflection:** Verwendet PHP’s `ReflectionClass` um die Eigenschaften des Objektes zu ermitteln und die entsprechenden Werte zu setzen oder auszulesen. Da dieser “Hydrator” von `$property->setAccessible(true)` gebraucht macht, können auf diesem Wege auch `private-Properties` bedient werden.

Und jetzt schauen wir uns das Ganze noch einmal unter Verwendung von “Fieldsets” an, denn auch da gibt es ein nützliches Feature, wenn man mit mehreren Objekten arbeitet, die sich untereinander referenzieren. Zunächst erstellen wir eine weitere Entity, die wir “`UserAddress`” nennen und in der wir die Anschrift des Users abbilden, die wir zukünftig im Rahmen der Anmeldung direkt mit Abfragen wollen. Die Daten sollen in der Anwendung aber eben eigenständig als Entity behandelt und auch nachher in der Datenbank in einer eigenen Tabelle gespeichert werden:

```
1  <?php
2  namespace Helloworld\Entity;
3
4  class UserAddress
5  {
6      private $street;
7      private $streetNumber;
8      private $zipcode;
9      private $city;
10
11     public function setStreet($street)
12     {
13         $this->street = $street;
14     }
15
16     public function getStreet()
17     {
18         return $this->street;
19     }
20
21     public function setCity($city)
22     {
23         $this->city = $city;
24     }
25
26     public function getCity()
27     {
28         return $this->city;
29     }
30
31     public function setStreetNumber($streetNumber)
32     {
33         $this->streetNumber = $streetNumber;
34     }
35
36     public function getStreetNumber()
37     {
38         return $this->streetNumber;
39     }
40
41     public function setZipcode($zipcode)
42     {
```

```
43         $this->zipcode = $zipcode;
44     }
45
46     public function getZipcode()
47     {
48         return $this->zipcode;
49     }
50 }
```

Listing 16.21

Die User-Entity erweitern wir um die Eigenschaft \$userAddress, die die Referenz zur passenden UserAddress-Entity symbolisiert:

```
1  <?php
2  namespace Helloworld\Entity;
3
4  class User
5  {
6      protected $id;
7      protected $email;
8      protected $name;
9      protected $userAddress;
10
11     public function setEmail($email)
12     {
13         $this->email = $email;
14     }
15
16     public function getEmail()
17     {
18         return $this->email;
19     }
20
21     public function setId($id)
22     {
23         $this->id = $id;
24     }
25
26     public function getId()
27     {
28         return $this->id;
29     }
```

```
30
31     public function setName($name)
32     {
33         $this->name = $name;
34     }
35
36     public function getName()
37     {
38         return $this->name;
39     }
40
41     public function setUserAddress($userAddress)
42     {
43         $this->userAddress = $userAddress;
44     }
45
46     public function getUserAddress()
47     {
48         return $this->userAddress;
49     }
50 }
```

Listing 16.22

Zusätzlich erstellen wir das neue `UserAddressFieldset` samt der erforderlichen “Filter-Spezifikation”, wie wir sie ja bereits kennengelernt haben, sowie dem Hinweis auf den zu verwendenden “Hydrator” und die passende Entity, in die die Daten dieses Fieldsets übernommen werden sollen:

```
1  <?php
2  namespace HelloWorld\Form;
3
4  use Zend\Form\Fieldset;
5
6  class UserAddressFieldset extends Fieldset
7  {
8      public function __construct()
9      {
10         parent::__construct('userAddress');
11         $this->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
12         $this->setObject(new \HelloWorld\Entity\UserAddress());
13
14         $this->add(array(
15             'name' => 'street',
```

```
16         'attributes' => array(
17             'type' => 'text',
18         ),
19         'options' => array(
20             'label' => 'Ihre Strasse:',
21         )
22     ));
23
24     $this->add(array(
25         'name' => 'streetNumber',
26         'attributes' => array(
27             'type' => 'text',
28         ),
29         'options' => array(
30             'label' => 'Ihre Hausnummer:',
31         )
32     ));
33
34     $this->add(array(
35         'name' => 'zipcode',
36         'attributes' => array(
37             'type' => 'text',
38         ),
39         'options' => array(
40             'label' => 'Ihre Postleitzahl:',
41         )
42     ));
43
44     $this->add(array(
45         'name' => 'city',
46         'attributes' => array(
47             'type' => 'text',
48         ),
49         'options' => array(
50             'label' => 'Ihre Stadt:',
51         )
52     ));
53     }
54 }
```

Listing 16.23

Das UserAddressFieldset binden wir entsprechend ein, allerdings nicht direkt in die SignUp-Form,

sondern “geschachtelt” in das UserFieldset:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Fieldset;
5
6  class UserFieldset extends Fieldset
7  {
8      public function __construct()
9      {
10         parent::__construct('user');
11
12         $this->add(array(
13             'name' => 'name',
14             'attributes' => array(
15                 'type' => 'text',
16                 'id' => 'name'
17             ),
18             'options' => array(
19                 'id' => 'name',
20                 'label' => 'Ihr Name:',
21             )
22         ));
23
24         $this->add(array(
25             'name' => 'email',
26             'attributes' => array(
27                 'type' => 'email',
28             ),
29             'options' => array(
30                 'id' => 'email',
31                 'label' => 'Ihre E-Mail-Adresse:'
32             ),
33         ));
34
35         $this->add(array(
36             'type' => 'Helloworld\Form\UserAddressFieldset',
37         )
38         );
39     }
40 }
```

Listing 16.24

Das eigentliche Formular sieht nun wie folgt aus:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Form;
5
6  class SignUp extends Form
7  {
8      public function __construct()
9      {
10         parent::__construct('signUp');
11         $this->setAttribute('action', '/signup');
12         $this->setAttribute('method', 'post');
13
14         $this->add(array(
15             'type' => 'Helloworld\Form\UserFieldset',
16             'options' => array(
17                 'use_as_base_fieldset' => true
18             )
19         ));
20
21         $this->add(array(
22             'name' => 'submit',
23             'attributes' => array(
24                 'type' => 'submit',
25                 'value' => 'Eintragen'
26             ),
27         ));
28     }
29 }
```

Listing 16.25

Wichtig ist hier das 'use_as_base_fieldset' => true, damit die Zuordnung von Werten und Objekteigenschaften korrekt funktioniert, die Daten ausgehend vom UserFieldset auf die entsprechenden Entities verteilt werden.

Zuletzt nicht vergessen, das View-File entsprechend der Fieldsets anzupassen, so dass die Felder auch alle korrekt angezeigt werden und es zu keinen Fehlern kommt:

```
1  <?php
2  $this->form->prepare();
3  echo $this->form()->openTag($this->form);
4
5  echo $this->formRow($this->form->get('user')
6      ->get('name'));
7
8  echo $this->formRow($this->form->get('user')
9      ->get('email'));
10
11 echo $this->formRow($this->form->get('user')
12     ->get('userAddress')->get('street'));
13
14 echo $this->formRow($this->form->get('user')
15     ->get('userAddress')->get('streetNumber'));
16
17 echo $this->formRow($this->form->get('user')
18     ->get('userAddress')->get('zipcode'));
19
20 echo $this->formRow($this->form->get('user')
21     ->get('userAddress')->get('city'));
22
23 echo $this->formSubmit($this->form->get('submit'));
24 echo $this->form()->closeTag();
```

Listing 16.26

Wenn wir nun das Formular mit gültigen Daten absenden, erhalten wir nicht mehr nur das User-Objekt, sondern auch direkt das gefüllte, referenzierte UserAddress-Objekt:

```
1  class Helloworld\Entity\User#201 (4) {
2      protected $id => NULL
3      protected $email => string(24) "zf2buch@michael-romer.de"
4      protected $name => string(13) "Michael Romer"
5      protected $userAddress =>
6          class Helloworld\Entity\UserAddress#190 (4) {
7              private $street => string(14) "Grevingstrasse"
8              private $streetNumber => string(2) "35"
9              private $zipcode => string(5) "48151"
10             private $city => string(8) "Münster"
11         }
12 }
```




Verarbeitung über Annotationen weiter vereinfachen

Der skizzierte Weg für die Verarbeitung von Formularen macht das Leben des Anwendungsentwicklers bereits deutlich leichter. Dennoch ist weiterhin recht viel Code notwendig, um die Konfigurationen vorzunehmen. Ein alternativer Weg ist die Verwendung von Annotationen in den Entity-Klassen, aus denen ein Großteil der andernfalls händisch erzeugten Konfiguration auch dynamisch vom Framework erzeugt werden kann. Weitere Informationen dazu finden sich bei Bedarf in der offiziellen Dokumentation des Frameworks.

17 Logging & Debugging

17.1 Einleitung

Spätestens dann, wenn irgendetwas nicht so funktioniert wie gedacht, muss man der Ursache für das Problem auf die Spur kommen. Für den Prozess des Debuggings, also des Ausmerzens von Fehlern, gibt es eine ganze Reihe verschiedener Vorgehen und Möglichkeiten der technischen Unterstützung. Häufig kann man stundenlang auf den Code schauen und sich trotzdem keinen Reim drauf machen, was da schief läuft. Da hilft es oftmals, wenn man den Code mal in Aktion betrachten und schauen kann, was da eigentlich tatsächlich passiert. Der Einsatz eines vollwertigen Debuggers wie etwa [XDebug](http://xdebug.org/)¹ kann dies leisten und lässt in Form und Funktion kaum Wünsche offen. Allerdings ist “XDebug” nach wie vor nicht ganz simpel zu installieren und mit der eigenen Entwicklungsumgebung zu verdrahten. Moderne IDEs, wie etwa [PhpStorm](http://www.jetbrains.com/phpstorm/)², bringen mittlerweile zwar schon eine sehr gute Unterstützung für das interaktive Debugging mit, dennoch ist und bleibt diese Disziplin immer ein wenig “schwarze Magie”.

Manchmal ist ein Debugger wie “XDebug” aber auch einfach zu viel des Guten oder in bestimmten Situationen auch gar nicht einsetzbar. So wirkt sich “XDebug” in produktiven Umgebungen spürbar negativ auf die Performance aus. In solchen Umgebungen, oder auch immer dann, wenn man nur mal kurz in den Wert der ein oder anderen Variable schauen will, kann man alternativ Debugging mit Hilfe von Logging betreiben. Dabei werden an definierten Stellen im Code bestimmte Werte aufgezeichnet und diese Aufzeichnungen im Nachgang dann vom Anwendungsentwickler ausgewertet. In diesem Fall betreibt man etwas, das auch als “Post-mortem debugging” bezeichnet wird: Man sieht die aufgezeichneten Werte erst im Nachhinein, nicht direkt zur Ausführungszeit des Scripts mit der Möglichkeit, die Verarbeitung zwischenzeitlich anzuhalten.

Zwei hilfreiche Logging-Tools, die sich direkt in den Browser integrieren und so sehr komfortabel während der Entwicklung zu nutzen sind, sind [FirePHP](http://www.firephp.org/)³, eine Erweiterung des beliebten [Firebug](http://getfirebug.com/)⁴ für den Mozilla Firefox, sowie [ChromePHP](http://www.chromephp.com/)⁵ für Google’s Chrome Browser. Firefox und Chrome sind derzeit die beiden mit sehr großem Abstand beliebtesten Browser in der Webentwickler-Gemeinde, nicht zuletzt genau wegen der Verfügbarkeit solcher Tools. Leider sprechen “FirePHP” und “ChromePHP” unterschiedliche Protokolle, so dass jeweils unterschiedliche Bibliotheken erforderlich sind, damit sie funktionieren. Beide Tools sind hervorragende Begleiter in der täglichen Entwicklung und der Beseitigung von Bugs.

Der Vollständigkeit halber und bevor wir in die Details der Tools schauen sei noch gesagt: Logging

¹<http://xdebug.org/>

²<http://www.jetbrains.com/phpstorm/>

³<http://www.firephp.org/>

⁴<http://getfirebug.com/>

⁵<http://www.chromephp.com/>

kann natürlich auch jenseits des Debuggings sehr hilfreich sein. Es kann das Mittel der Wahl für das Debugging sein, es kann aber auch in ganzen Situationen eingesetzt werden, klar.



Anzeige von Fehlern aktivieren

Während der Entwicklung ist es sinnvoll, über die `php.ini` die Anzeige von Fehlern zu aktivieren. Andernfalls sieht man oftmals nur einen weißen Bildschirm und muss sich die Fehlermeldung ggf. mühsam aus den Logfiles des Webserverns heraussuchen. Denn nicht immer werden etwa Exceptions vom Framework “gefangen”, so z.B. dann nicht, wenn zu einer Action das entsprechende Template nicht gefunden werden kann: “Fatal error: Uncaught exception ‘Zend\View\Exception\RuntimeException’ with message ‘Zend\View\Renderer\PhpRenderer::render: Unable to render template “helloworld/auth/login”; resolver could not resolve to a file”.

Die Anzeige von Fehlern lässt sich in der `php.ini` über die Parameter `display_startup_errors` und `display_errors` steuern, die man in einer Entwicklungsumgebung beide aktivieren sollte. Alternativ lassen sich die Einstellungen auch über die PHP-Funktion `ini_set` mit `ini_set('display_startup_errors', true);` und `ini_set('display_errors', true);` zur Laufzeit setzen. Allerdings hilft das nur, wenn der Fehler auch dann zeitlich auftritt, nach dem ebenjene Codezeilen in der Anwendung bereits ausgeführt wurden.

17.2 FirePHP für Firefox

Für den Einsatz von “FirePHP” ist neben einer Firefox-Installation samt [Firebug](#)⁶ mit [Firebug-FirePHP-Erweiterung](#)⁷ noch die entsprechende PHP-Bibliothek in der eigenen Anwendung erforderlich, die sich am einfachsten wieder über “Composer” installieren lässt.

```
1 "require": {  
2     "firephp/firephp-core": "dev-master"  
3 }
```

Listing 17.1

Nach einem Update der Abhängigkeiten durch “Composer” via

```
1 $ php composer.phar update
```

im Projektverzeichnis kann “FirePHP”, etwa in einem ActionController, bereits verwendet werden:

⁶<https://getfirebug.com/>

⁷<http://www.firephp.org/>

```
1 <?php
2 FB::log('Log message');
3 FB::info('Info message');
4 FB::warn('Warn message');
5 FB::error('Error message');
```

Listing 17.2

“FirePHP” stellt für die unterschiedlichen Log-Level, die den Typ des jeweiligen Logeintrags definierten, eigene, selbsterklärende Methoden zur Verfügung. Die geloggten Werte von “FirePHP” integrieren sich in die bereits vorhandenen Ansichten von “Firebug”. Je nach Log-Level sieht man in der Konsole die entsprechende Darstellung. Auch lässt sich zusätzlich zum eigentlichen Wert noch ein Label angeben, dass dann mit in der Konsole angezeigt wird:

```
1 <?php
2 //[...]
3 \FB::log($page->getVariable('greeting'), 'label');
4 //[...]
```

Listing 17.3

Die Daten von “FirePHP” (die Server-Komponente ist hier gemeint) werden übrigens mit Hilfe sog. “X-Header” übertragen, also durch nicht-standardisierte Zusatzwerte als Teil des HTTP-Headers. Ein wirklich unermesslicher Vorteil dieser Form des Loggings ist daher die Tatsache, dass der Output nicht die eigentliche Darstellung der Seite berührt, wie das ja der Fall ist, wenn man sich `var_dump()`, `print_r()` oder ähnlichem bedient.

17.3 ChromePHP für Chrome

Die Installation von “ChromePHP” muss zum Zeitpunkt des Schreibens dieses Kapitels noch manuell erledigt werden, ist aber schnell gemacht, da “ChromePHP” nur eine einzige PHP-Klasse mitbringt. Ist “ChromePHP” heruntergeladen, muss man nur noch dafür sorgen, dass die jeweilige Klasse verfügbar ist. Ich inkludiere sie der Einfachheit halber einmal im Rahmen der `init_autoloader.php`:

```
1 <?php
2 include 'vendor/chrome/php/Chrome.php';
```

Listing 17.4

Zusätzlich zur PHP-Bibliothek muss auch der Chrome-Browser noch mit der entsprechenden [ChromePHP-Extension](https://chrome.google.com/webstore/detail/noaneddfkdjfnfdakjjmocnfnkfehhd)⁸ ausgestattet werden.

⁸<https://chrome.google.com/webstore/detail/noaneddfkdjfnfdakjjmocnfnkfehhd>

Im Anschluss können analog zu “FirePHP” Werte geloggt werden, die dann im Chrome in der Konsole des Browsers (im Gegensatz zu “FirePHP” ist hier kein “Host-Plugin” wie “Firebug” erforderlich) zu sehen sind:

```
1 <?php
2 \ChromePhp::log($page->getVariable('greeting'));
```

Listing 17.5

Auch bei “ChromePHP” kann zusätzlich ein Label für das jeweilige Datum angegeben werden, das in der Anzeige verwendet wird:

```
1 <?php
2 \ChromePhp::log('label', $page->getVariable('greeting'));
```

Listing 17.6

Allerdings wird beim “ChromePHP” zunächst das “Label” angegeben und im Anschluss der eigentliche Wert. Bei “FirePHP” ist dies andersherum.

17.4 Zend\Log

Wir haben mit “FirePHP” und “ChromePHP” jetzt zwei Tools für das Logging während der Entwicklung der Anwendung kennengelernt. Im echten Betrieb wird man ein weiteres Tool einsetzen wollen, das die geloggtten Werten in irgendeiner Form speichert, etwa in einer Datenbank, Logfiles oder ähnlichem, so dass sie persistiert und vom Anwendungsentwickler, der ja im Produktivbetrieb nicht mehr auch gleichzeitig der Nutzer ist, später eingesehen werden können.

Abgesehen davon haben alleine bereits “FirePHP” und “ChromePHP”, obwohl sie nahezu das gleiche tun, eine unterschiedliche API, bieten also leicht unterschiedliche Methodenaufrufe für den Anwendungsentwickler. Will man beide Tools einsetzen, hat man eigentlich schon ein Problem.

Hier kommt Zend\Log in Spiel, die Logging-Komponente des Zend Framework 2, die es in einer ähnlichen Form auch bereits in der Version 1 gegeben hat. Zend\Log hilft bei dem oben genannten Problem dahingehend, dass es zwischen dem sog. Logger, dem Objekt, der den log()-Aufruf entgegen nimmt, und dem Writer, der dafür sorgt, dass das zu loggende auch irgendwo hingeschrieben wird, unterscheidet. Je nach Situation und Laufzeitumgebung kann der Writer ausgetauscht werden, ohne, dass der Logging-Aufruf log() im Code modifiziert oder dupliziert werden müsste. Es lassen sich sogar gleich mehrere Writer auf einmal an einen Logger hängen, die geloggte Nachricht an mehrere Stellen gleichzeitig schreiben.

17.5 Zend\Log mit FirePHP

Praktisch ist, dass Zend\Log direkt die Implementierung für den “FirePHP”-Writer mitbringt, dass ganze also “Out-of-the-Box” - funktioniert. Die folgenden Zeilen Code reichen aus, um mit “FirePHP” via Zend\Log zu loggen:

```
1  <?php
2  $logger = new \Zend\Log\Logger();
3  $logger->addWriter('FirePhp');
4  $logger->log(\Zend\Log\Logger::INFO, $page->getVariable('greeting'));
```

Listing 17.7

Wichtig ist, dass Zend\Log mit dem FirePhpWriter keine eigene Implementierung des “FirePHP”-Protokolls mitbringt, sondern nur den “Glue Code” zwischen Zend\Log und der “FirePHP”-Bibliothek. Sprich, man muss nach wie vor zunächst dafür sorgen, dass “FirePHP” zur Verfügung steht. Erst dann funktioniert das Logging mit “FirePHP” via Zend\Log.

Der Aufruf

```
1  <?php
2  $logger->addWriter('FirePhp');
```

Listing 17.8

funktioniert übrigens deshalb, weil der WriterPluginManager von Zend\Log einige Schlüsselworte definiert und auf die jeweiligen Writer-Implementierungen, die standardmäßig mitgeliefert werden, abbildet:

```
1  <?php
2  // [...]
3  protected $invokableClasses = array(
4      'db'          => 'Zend\Log\Writer\Db',
5      'firephp'     => 'Zend\Log\Writer\FirePhp',
6      'mail'        => 'Zend\Log\Writer\Mail',
7      'mock'        => 'Zend\Log\Writer\Mock',
8      'null'        => 'Zend\Log\Writer\Null',
9      'stream'      => 'Zend\Log\Writer\Stream',
10     'syslog'       => 'Zend\Log\Writer\Syslog',
11     'zendmonitor' => 'Zend\Log\Writer\ZendMonitor',
12 );
13 // [...]
```

Listing 17.9

Die Variablenbezeichnung `$invokableClasses` verrät es bereits: Der `WriterPluginManager` ist wieder ein sog. “Service Manager”, kann also wie der `ServiceManager` der Anwendung, der sich als zentraler Manager aller “Anwendungsservices” des Systems versteht, nicht nur `Writer` auf Basis einer Klasse erzeugen, sondern etwa auch eine “Factory” dafür verwenden.

17.6 Zend\Log mit ChromePHP

Um “ChromePHP” mit `Zend\Log` verwenden zu können, muss dem zunächst ein entsprechender `Writer` zur Verfügung gestellt werden, denn leider kennt der `WriterPluginManager`, im Gegensatz zu `FirePHP`, das entsprechende Tool leider nicht.

Ein eigener `Writer` ist schnell geschrieben. Der Einfachheit legen wir die entsprechende Klasse in unserem `HelloWorld`-Modul in der Datei `src/HelloWorld/Log/Writer/ChromePhp.php` ab:

```
1  <?php
2  namespace HelloWorld\Log\Writer;
3
4  use Zend\Log\Writer\AbstractWriter;
5  use Zend\Log\Formatter\FirePhp as FirePhpFormatter;
6
7  class ChromePhp extends AbstractWriter
8  {
9      public function __construct ()
10     {
11         $this->formatter = new FirePhpFormatter();
12     }
13
14     protected function doWrite(array $event)
15     {
16         $line = $this->formatter->format($event);
17
18         switch (strtolower($event['priorityName'])) {
19             case 'error':
20                 \ChromePhp::error($line);
21                 break;
22             case 'warn':
23                 \ChromePhp::warn($line);
24                 break;
25             default:
26                 \ChromePhp::log($line);
```

```
27         }
28     }
29 }
```

Listing 17.10

Auch hier muss “ChromePHP” im Vorfeld wieder verfügbar gemacht werden, damit der `Writer`, der als “Glue Code” zwischen `Zend\Log` und der “ChromePHP”-Bibliothek fungiert, seinen Dienst verrichtet.

Wirklich interessant ist eigentlich nur die Methode `doWrite()`, in der das eigentliche, “ChromePHP”-spezifische Logging implementiert ist. Der eingesetzte Formatter sorgt übrigens dafür, dass die Nachricht so formatiert geloggt wird, wie man es gerne hätte bzw. es jeweils notwendig oder sinnvoll ist. In diesem Fall habe ich es mir leicht gemacht und einfach den vom Framework mitgelieferten `FirePhpFormatter` verwendet, weil er ganz ähnliche Ausgaben wie die produziert, die man sich von “ChromePHP” wünscht. Auch der Formatter lässt sich natürlich nach Belieben für das Logging mit “ChromePHP” anpassen, falls der Bedarf dafür besteht.

Nun können wir an Ort und Stelle des Geschehens wie folgt mit “ChromePHP” loggen:

```
1      <?php
2      $logger = new \Zend\Log\Logger();
3      $logger->addWriter(new \HelloWorld\Log\Writer\ChromePhp());
4
5      $logger->log(
6          \Zend\Log\Logger::INFO,
7          $page->getVariable('greeting')
8      );
```

Listing 17.11

17.7 Zend\Log situationsabhängig einsetzen

Noch sind wir allerdings kein wirkliches Stück weiter, weil wir immer gleich an Ort und Stelle des Loggings auch bereits schon das “Wie?” beantworten. Wir sollten stattdessen die Erzeugung und Konfiguration des Loggers im Rahmen des Moduls erledigen und den fertigen Logger jeweils nur am “Logging-Geschehen” über den `ServiceManager` beziehen. Dazu könnten wir die `module.config.php` des `HelloWorld`-Moduls im Abschnitt `service_manager` wie folgt einrichten:


```
1  <?php
2  // [...]
3  'service_manager' => array(
4      'factories' => array(
5          'logger' => function($sl)
6              {
7                  $logger = new \Zend\Log\Logger();
8                  $logger->addWriter(new \Helloworld\Log\Writer\ChromePhp());
9                  return $logger;
10             }
11     )
12 )
13 // [...]
```

Listing 17.12

und dann an Ort und Stelle nur noch

```
1  <?php
2  $this->getServiceLocator()
3      ->get('logger')
4      ->log(\Zend\Log\Logger::INFO, $page->getVariable('greeting'));
```

Listing 17.13

aufrufen. Um den verwendeten Logger jetzt noch konfigurabel zu machen, können wir einen Abschnitt logger in die module.config.php hinzufügen

```
1  <?php
2  // [...]
3  'logger' => array(
4      'writer' => 'ChromePhp'
5  )
6  // [...]
```

Listing 17.14

auf das wir in der Logger-Factory zugreifen:

```

1  <?php
2  // [...]
3  'service_manager' => array(
4      'factories' => array(
5          'logger' => function($sl) {
6              $logger = new \Zend\Log\Logger();
7              $config = $sl->get('Config');
8
9              if ($config['logger']['writer'] == 'ChromePhp') {
10                 $logger->addWriter(
11                     new \Helloworld\Log\Writer\ChromePhp()
12                 );
13             } else {
14                 $logger->addWriter('FirePhp');
15             }
16
17             return $logger;
18         }
19     )
20 )
21 // [...]

```

Listing 17.15

Die Implementierung der Factory ist sicherlich nicht optimal, keine Frage. Aber an dieser Stelle soll es mal so ausreichen, geht es uns doch um das Prinzip: Über den ServiceManager, der der Factory vom Framework übergeben wird, können wir auf die entsprechende Konfiguration zugreifen und den Logger mit dem gewünschten Writer ausstatten.

Jetzt haben wir es fast. Damit die beiden Entwickler, von denen der einen mit “FirePHP” arbeitet und der andere “ChromePHP” bevorzugt, jetzt aber nicht ständig die `module.config.php` bearbeiten und ggf. Werte verändern, die im Produktivsystem unbedingt beibehalten werden müssen, legen sie sich jeweils eine individuelle Config-Datei, etwa `dev1.local.php` im Verzeichnis `/config/autoload` der Anwendung an. Dort legen sie ihre persönliche Konfiguration für das Logging fest und überschreiben damit die Konfiguration der `module.config.php`:

```

1  <?php
2  return array(
3      'logger' => array(
4          'writer' => 'FirePhp'
5      )
6  );

```

Listing 17.16

Und schon wird auf dem jeweiligen System der `Writer` der Wahl eingesetzt. Die Datei `dev1.local.php` (bzw. `dev2.local.php`) checken die Entwickler dann entsprechend auch nicht in das Code-Repository ein.

Etwas unangenehm ist jetzt allerdings noch die Tatsache, dass der Aufruf für das Logging, etwa in einem Controller, recht umständlich ist:

```
1 <?php
2 $this->getServiceLocator()
3     ->get('logger')->log(\Zend\Log\Logger::INFO, 'message');
```

Listing 17.17

Schön wäre es, wenn wir es zumindest auf

```
1 <?php
2 $this->logger->log(\Zend\Log\Logger::INFO, 'message');
```

Listing 17.18

kürzen könnten. Es sind nur ein paar Zeichen, aber gelogged wird ja vermutlich häufig, also lohnt es sich und außerdem haben wir dann eine Abhängigkeit - nämlich die zum `ServiceLocator`, zumindest mal beim Logging - eliminiert.

Weil wir den Logger in jedem Controller unserer Anwendung auf diese Art verfügbar machen wollen, bietet es sich an, ihn standardmäßig in Form eines Initializer zu injizieren. Dazu registrieren wir im Abschnitt `controllers` in der `module.config.php` den entsprechenden Initializer, in diesem Beispiel in Form einer Callback-Funktion:

```
1 <?php
2 // [...]
3 'controllers' => array(
4     // [...]
5     'initializers' => array(
6         'logger' => function($instance, $serviceManager){
7             if ($instance instanceof \Helloworld\Controller\LoggerAware) {
8                 $instance->setLogger(
9                     $serviceManager
10                        ->getServiceLocator()
11                        ->get('logger')
12                    );
13             }
14         }
15     )
16 )
```

Listing 17.19

Wie war das noch mal mit dem Initializer? Alle bei einem “ServiceManager” registrierten Initializer werden nach der Erzeugung eines Services aufgerufen und bekommen die erzeugte Instanz und den jeweiligen “ServiceManager” selbst übergeben. Es ist dann Aufgabe des Initializer, festzustellen, ob es für ihn etwas zu tun gibt, oder eben nicht. Dazu arbeitet man am besten mit speziellen “Aware”-Interfaces, die dem Initializer signalisieren, ob er aktiv werden muss oder nicht:

```
1  <?php
2  namespace Helloworld\Controller;
3
4  interface LoggerAware
5  {
6      public function getLogger();
7      public function setLogger(\Zend\Log\Logger $logger);
8  }
```

Listing 17.20

Jeder Controller, der das LoggerAware-Interface implementiert, bekommt den Logger nun automatisch injiziert:

```
1  <?php
2  namespace Helloworld\Controller;
3
4  // use [...]
5
6  class IndexController
7      extends ActionController
8      implements LoggerAware
9  {
10     // [...]
11 }
```

Listing 17.21

Eine kurze Notiz am Rande: Wichtig ist hier, dass wir den Initializer im ControllerManager hinterlegen, der ja selbst einen “ServiceManager” darstellt. Würden wir den Initializer an den “ServiceManager für die zentralen Anwendungsservices hängen, würde der Initializer nicht ausgeführt werden, weil er den Service nicht selbst erzeugt, sondern sich stattdessen eines seiner “Sub-ServiceManager”, eben dem ControllerManager bedient hat.

17.8 Weitere Möglichkeiten mit Zend\Log

Formatter

Die Möglichkeit, spezielle Formatter für die zu loggenden Nachrichten einzusetzen, hatte ich bereits erwähnt. So bietet `Zend\Log\Formatter\Xml` etwa die Möglichkeit, ein XML-Schnippselchen zu erzeugen, falls es für das Logging erforderlich ist.

Filter

Mit Hilfe eines `Filter` lässt sich für einen `Writer` definieren, auf welche “Log-Priorität” er anspringen soll. `Zend_Log` kennt die folgenden “Log-Prioritäten”, auch “Log-Level” genannt:

- `EMERG = 0`; // GAU: Das System ist defekt
- `ALERT = 1`; // Notfall: Es besteht dringender Handlungsbedarf
- `CRIT = 2`; // Kritische Situation
- `ERR = 3`; // Error
- `WARN = 4`; // Warnung
- `NOTICE = 5`; // Hinweis
- `INFO = 6`; // Info
- `DEBUG = 7`; // Debugging

So ließe sich auf diesem Wege im Produktivsystem standmäßig ein “SMS-Writer” an den Logger hängen, der allerdings immer nur dann aktiv wird, wenn es sich um einen Log-Eintrag mit Level “EMERG” oder “ALERT” handelt.

Filter können aber nicht nur auf das “Log-Level” hin ausgerichtet werden, sondern via `Regex` auch auf bestimmte Lognachrichten. Die Möglichkeiten, das Logging je nach Situation und Schweregrad zu konfigurieren, sind also vielfältig.

18 Unit Testing

18.1 Einleitung

Für mich ist ein gutes Webframework nur so gut, wie es das Unit Testing ermöglicht. Ich denke, Unit Testing ist unabdingbar für Agilität und Agilität wiederum ist unabdingbar für ein dauerhaft konkurrenzfähiges Produkt, dass sich kurzfristig an neue Gegebenheiten und Anforderungen anpassen kann, ohne, dass durch Änderungen die Integrität des Bestehenden torpediert wird. Dabei spielt Unit Testing eine wirklich immens wichtige Rolle. Ein Framework muss es dem Anwendungsentwickler leicht machen, einzelne Bestandteile separat vom Rest zu testen und automatisch auf die geplante Funktionsweise hin abzuklopfen. Und das so gut, dass ich, nach dem ich irgendeine Änderung am System vorgenommen habe, jederzeit durch Ausführung meiner Tests, sicher sein kann, dass ich nicht hier etwas angepasst habe und dort etwas umgefallen ist. Nur so kann ich mir sicher sein, dass die Änderung keine negativen Seiteneffekte hatte, die mir vielleicht verborgen geblieben sind, die erst im Produktivbetrieb zum Vorschein kommen und viel größere Probleme und Aufwände erzeugen, als die, die zur Erstellung des Tests notwendig gewesen wären.

Um Unit Testing effektiv zu ermöglichen, müssen die einzelnen, zu testenden Bestandteile der Anwendung, so miteinander verbunden sein, dass man sie jederzeit problemlos auch wieder entflechten und ggf. isoliert testen oder in einer anderen Art und Weise für Tests wieder zusammenstecken kann. Das Zend Framework 2 bietet die besten Voraussetzungen dafür. Neben dem Modulsystem ist gerade in meinen Augen diese Tatsache das Argument für das Framework schlechthin. War es in Version 1 noch unangenehm bis unmöglich, vernünftig etwa Controller-Klassen zu testen, so ist dies in der neuen Version so einfach, wie es nur sein kann. Dies manifestiert sich etwa in der Tatsache, dass es die Komponente `Zend_Test` aus Version 1 nun schlichtweg nicht mehr gibt, weil man sie auch nicht mehr braucht. Praktisch benötigt man nun nicht mehr als "PHPUnit" (oder eine der Alternativen), die Basis für das Unit Testing. Aber nicht nur das: Insbesondere auch der Ansatz des "Dependency Injection", der sich im Zend Framework 2 über den `ServiceManager` und `Zend\Di` manifestiert, leistet enorme Schützenhilfe. Denn häufig machen es hart verdrahtete Abhängigkeiten zu anderen Objekten so schwierig, einzelne Funktionen isoliert zu testen. Abhängigkeiten zu "echten" Services, die im normalen Betrieb notwendig sind, können so im Rahmen von Tests leicht aufgelöst werden, indem stattdessen "Fake Objekte" injiziert werden. Durch diese gute Ausgangssituation steht dem Unit Testing mit dem Zend Framework 2, ggf. auch im Rahmen eines Test-Driven Developments, nichts im Wege.

Vorab noch kurz ein paar Hinweise zur Terminologie: Überblicherweise wird im Rahmen von Unit Testing und "Fake Objects" auch von "Mocks" und "Stubs" gesprochen. Die Definitionen gehen da leider mittlerweile weit auseinander, aber für mich und die meisten Autoren ist ein "Fake Object" jedes Objekt, das ein real existierendes Objekt (im Rahmen eines Tests) ersetzt. Ein "Mock" ist ein "Fake Object", das dazu beiträgt, festzustellen, ob ein Test fehlgeschlagen ist. Ein "Stub" ist ein "Fake

Object” im Sinne einer Nachbildung eines existierenden Objekts, um eine Funktion zu emulieren, nicht aber dabei zu helfen, den Erfolg oder den Fehlschlag eines Tests zu validieren. Konkretes dazu aber auch noch im Folgenden. Dann wird es vermutlich klarer.

Für die folgenden Abschnitte ist erforderlich, dass du über ein grundsätzliches Verständnis für die Funktionsweise von PHPUnit verfügst und zudem “PHPUnit” auf dem System, auf dem du die Tests ausführen willst (ggf. zunächst erst einmal das System, mit dem du auch entwickelst), installiert und einsatzbereit ist.

18.2 Einen Service testen

Ganz zu Anfang dieses Buches haben wir den GreetingService erstellt, der uns abhängig von der Tageszeit die passende Begrüßung liefert. Der Service ist sehr überschaubar und daher ein guter Start in dieses Kapitel:

```
1  <?php
2  namespace Helloworld\Service;
3
4  class GreetingService
5  {
6      public function getGreeting()
7      {
8          if(date("H") <= 11)
9              return "Good morning, world!";
10         else if (date("H") > 11 && date("H") < 17)
11             return "Hello, world!";
12         else
13             return "Good evening, world!";
14     }
15 }
```

Listing 18.1

Den GreetingService stellen wir den anderen Objekten der Anwendung im normalen Betrieb über den ServiceManager zur Verfügung. Für den Test können auf den ServiceManager verzichten und den Service direkt instanzieren, die notwendigen Tests durchführen und schauen, ob alles wie geplant funktioniert.

Ich hatte seinerzeit schon angekündigt, dass der Einsatz der PHP-Funktion `date()` uns bei Tests das Leben noch schwer machen wird. Warum? Weil wir aktuell keine Möglichkeit haben, das Ergebnis von `date()` für die Tests zu manipulieren. Um alle 3 Fälle zu Testen, müssten wir den Tests also mindestens mal zu 3 verschiedenen Tageszeiten, eigentlich aber einmal zu jeder volle Stunde, laufen lassen, um jeden Fall einmal in Aktion sehen zu können. Das ist natürlich kein Zustand. Klar,

wir könnten im Rahmen des Testfalls natürlich auch jeweils die Systemzeit manipulieren, aber das ist wohl erst recht mal kein Zustand und hätte im Zweifelsfall auch wieder Seiteneffekte, die wir nicht kontrollieren können. Zunächst einmal sollten wir also die Abhängigkeit zur `date()`-Funktion auflösen, die uns das Testen schwer macht.

Im ersten Schritt erstellen wir dazu ein neues Interface, dass wir in `src/Helloworld/Service/GreetingService/Hou` definieren:

```
1  <?php
2  namespace Helloworld\Service\GreetingService;
3
4  interface HourProviderInterface
5  {
6      public function getHour();
7  }
```

Listing 18.2

Mit dem `HourProviderInterface` haben wir nun die Anforderung an einen “HourProvider” definiert, eine Komponente, die uns die aktuelle Stunde des Tages liefert. Eine erste Implementierung dieses Interfaces ist ebenjener “HourProvider”, der sich der `date()`-Funktion bedient:

```
1  <?php
2  namespace Helloworld\Service\GreetingService;
3
4  use Helloworld\Service\GreetingService\HourProviderInterface;
5
6  class HourProviderDateFunction implements HourProviderInterface
7  {
8      public function getHour()
9      {
10         return date("H");
11     }
12 }
```

Listing 18.3

Wenn wir in der `GreetingServiceFactory` im Produktivbetrieb nun dafür sorgen, dass diese konkrete `HourProviderDateFunction` injiziert wird


```
1  <?php
2  namespace HelloWorld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class GreetingServiceFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $greetingService = new GreetingService();
12
13         $hourProvider =
14             new GreetingService\HourProviderDateFunction();
15
16         $greetingService->setHourProvider($hourProvider);
17         return $greetingService;
18     }
19 }
```

Listing 18.4

so haben wir weiterhin die richtige Ausgabe auf dem Bildschirm, z.B. wenn wir den Service etwa im Rahmen eines Controllers verwenden (und dessen Output per `var_dump()` sichtbar machen):

```
1  string(13) "Hello, world!"
```

Für mich ist gerade 16:32 Uhr, daher passt die Ausgabe soweit weiterhin, auch nach der Auslagerung des “Hour Provider”. Natürlich vorher nicht vergessen, den `ServiceManager` auf die entsprechende Factory hinzuweisen, so dass das `HourProviderDateFunction`-Objekt auch bereitsteht. Aber das muss ich jetzt mittlerweile ja vermutlich schon gar nicht mehr explizit erwähnen. Außerdem muss der `GreetingService` natürlich über die notwendige Eigenschaft samt “Getter” und “Setter” verfügen:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  use HelloWorld\Service\GreetingService\HourProviderInterface;
5
6  class GreetingService
7  {
8      private $hourProvider;
9
10     public function getGreeting()
11     {
12         if (!$this->hourProvider)
13             throw new \BadMethodCallException('HourProvider not yet set.');
```

```
14
15         $hour = $this->hourProvider->getHour();
16
17         if($hour <= 11)
18             return "Good morning, world!";
19         else if ($hour > 11 && $hour < 17)
20             return "Hello, world!";
21         else
22             return "Good evening, world!";
23     }
24
25     public function setHourProvider(HourProviderInterface $hourProvider)
26     {
27         $this->hourProvider = $hourProvider;
28     }
29
30     public function getHourProvider()
31     {
32         return $this->hourProvider;
33     }
34 }
```

Listing 18.4

So, nun aber zum eigentlichen Test. Bislang waren das alles ja nur die Vorarbeiten, um das Testen überhaupt erst zu ermöglichen. Gerade auch bei gewachsenen Anwendungen ist das Problem mit dem Unit Testing ja meist gar nicht, dass man nicht testen will (das ist ja oft auch gar nicht schwer), sondern, dass man es schlichtweg nicht kann. Da heißt es dann also zunächst mal Refactoring zu betreiben, so wie wir es gerade getan haben.

Jetzt erstellen wir einen alternativen “HourProvider”, den wir im Rahmen des Tests später verwenden und den wir dann auch vollständig kontrollieren können:

```
1  <?php
2  namespace HelloWorldTest\ServiceTest\GreetingServiceTest;
3
4  use HelloWorld\Service\GreetingService\HourProviderInterface;
5
6  class HourProviderFake implements HourProviderInterface
7  {
8      private $hourToReturn = 0;
9
10     public function getHour()
11     {
12         return $this->hourToReturn;
13     }
14
15     public function setHourToReturn($hourToReturn)
16     {
17         $this->hourToReturn = $hourToReturn;
18     }
19
20     public function getHourToReturn()
21     {
22         return $this->hourToReturn;
23     }
24 }
```

Listing 18.5

Weil wir den HourProviderFake nur für Tests verwenden, legen wir ihn in der Datei /tests/HelloWorldTest/ServiceTest/GreetingServiceTest.php ab und erstellen damit auch im Projektverzeichnis zunächst den “Top-Level-Folder” tests, sowie HelloWorldTest für alle Tests rund um das HelloWorld-Modul, sowie ServiceTest für alle Service-Tests des HelloWorld-Moduls. So bewahren wir uns auch später noch die Übersicht, wenn der Testumfang mal größer wird.

Den HourProviderFake verwenden wir jetzt im Rahmen unseres ersten Tests auf Basis von “PHPUnit”. Die folgende Testklasse legen wir in der Datei /tests/HelloWorldTest/ServiceTest/GreetingServiceTest.php ab:

```
1  <?php
2  namespace HelloworldTest\ServiceTest;
3
4  class GreetingServiceTest extends \PHPUnit_Framework_TestCase
5  {
6      public function testGetGreeting()
7      {
8          $greetingService = new \Helloworld\Service\GreetingService();
9          $hourProviderFake = new GreetingServiceTest\HourProviderFake();
10         $greetingService->setHourProvider($hourProviderFake);
11
12         for($i = 0; $i <= 24; $i++)
13         {
14             $hourProviderFake->setHourToReturn($i);
15             $greeting = $greetingService->getGreeting();
16
17             if($i <= 11)
18                 $this->assertEquals("Good morning, world!", $greeting);
19             else if ($i > 11 && $i < 17)
20                 $this->assertEquals("Hello, world!", $greeting);
21             else
22                 $this->assertEquals("Good evening, world!", $greeting);
23         }
24     }
25 }
```

Listing 18.6

Wie gesagt erzeugen wir im Rahmen des Tests den GreetingService “von Hand” und nicht wie im Produktivsystem über den ServiceManager. Zudem statten wir den GreetingService mit dem HourProviderFake aus. Danach testen wir für jede Stunde des Tages, ob das Ergebnis dem entspricht, was erwartet wird.

Zusätzlich zur eigentlichen Testklasse benötigen wir noch die phpunit.xml, über die wir die auszuführenden Tests definieren. Dies hat den Vorteil, dass wir später mit einem Kommando alle Tests ausführen können:

```

1 <phpunit bootstrap="./bootstrap.php">
2     <testsuites>
3         <testsuite name="AllTests">
4             <directory>./HelloworldTest/ServiceTest</directory>
5         </testsuite>
6     </testsuites>
7 </phpunit>

```

Jetzt haben wir es fast. Was noch fehlt, ist das Bootstrapping der Anwendung im Rahmen von Tests:

```

1 <?php
2 use Zend\Loader\StandardAutoloader;
3
4 chdir(dirname(__DIR__));
5
6 include 'init_autoloader.php';
7
8 $loader = new StandardAutoloader();
9 $loader->registerNamespace('HelloworldTest', __DIR__ . '/HelloworldTest');
10 $loader->register();
11
12 Zend\Mvc\Application::init(include 'config/application.config.php');

```

Listing 18.7

Im Grunde genommen könnten wir an dieser Stelle auch noch darauf verzichten, das Framework überhaupt in irgendeiner Form “zu belästigen”, testen wir bislang doch nur sog. “POPOs” (Plain Old PHP Objects) und benötigen direkt keinerlei Framework-Klassen. Allerdings wird ja etwa das Autoloading der Modul-Klassen, also z.B. die von `Helloworld\Service\GreetingService`, über das Zend Framework 2 realisiert. Würden wir `Zend\Mvc\Application` hier also noch ganz Außen vorlassen, wären einige Probleme vorprogrammiert

```

1 $ phpunit
2 > PHP Fatal error:  Class 'Helloworld\Service\GreetingService'
3 > not found in
4 > /vagrant/tests/HelloworldTest/ServiceTest/GreetingServiceTest.php
5 > on line 8

```

und wir müssten uns selbst um das Autoloading der Klassen kümmern bzw. mit ständigen `require()`-Aufrufen unsere Testklassen aufblähen. Die weiteren Statements der `bootstrap.php` zielen ebenfalls auf das Autoloading von PHP-Klassen ab: Einmal auf die des Framework selbst, sowie auf die Testklassen, die es zu entwickeln gilt. Die `init_autoloader.php` hatten wir ja bereits vorher schon in der Anwendung. Sie kam durch die “`ZendSkeletonApplication`”.

Der `test`-Folder der Anwendung hat nun also den folgenden Aufbau:

```
1 tests/
2     HelloWorldTest/
3         ServiceTest/
4             GreetingServiceTest/
5                 HourProviderFake.php
6             GreetingServiceTest.php
7     bootstrap.php
8     phpunit.xml
```

Wenn wir jetzt vom Projektverzeichnis aus in den tests-Folder wechseln und dort das Kommando `phpunit` ausführen, sollte diese oder eine sehr ähnliche Ausgabe auf dem Bildschirm zu sehen sein:

```
1 $ phpunit
2 > PHPUnit 3.6.12 by Sebastian Bergmann.
3 > Configuration read from /vagrant/tests/phpunit.xml
4 > .
5 > Time: 0 seconds, Memory: 9.75Mb
6 > OK (1 test, 25 assertions)
```

Sehr schön, der Test verläuft positiv. Halten wir fest: Wir können unsere Services testbar gestalten, indem wir Abhängigkeiten so realisieren, dass wir zur Laufzeit die notwendigen “echten” Kollaborateure injizieren (etwa über eigene Factories oder `Zend\Di`) und für Tests die entsprechenden Fake-Objekte verwenden. Dazu empfiehlt es sich, immer zunächst über ein Interface die Anforderungen an den Kollaborateur zu definieren, auf dessen Basis sowohl den “echten” Service zu erstellen (ggf. später auch mal alternative (bessere?) Implementierungen), sowie die, im Rahmen von Tests benötigten Fake-Objekte. Wenn man das konsequent durchzieht, werden Tests - und zwar nicht nur die von Services - wirklich zu einem Kinderspiel.

18.3 Einen Controller testen

Einen Controller zu testen ist mit dem Zend Framework 2 keine unlösbare Aufgabe. Natürlich kann man auch seine Controller so schreiben, dass sie faktisch nicht mehr testbar sind, allerdings steuert das Framework durch seine Struktur und die Tatsache, dass Controller in Version 2 von ihrer Art her “nichts besonderes” mehr sind, massiv dagegen. Macht man auch für seine Controller wie schon bei den Services gebraucht von “Dependency Injection” und versucht grundsätzlich den Code der einzelnen Actions so überschaubar wie möglich zu halten, stehen die Vorzeichen beim Testen auch für Controller und Actions gut. Gerade in Version 1 des Frameworks waren die ja nicht ganz unwichtigen Controller auf Grund ihrer Struktur und der mangelnden “Service-Orientierung” meist die “untestbaren” Komponenten eines Systems.

Schauen wir uns zunächst einen einfachen Fall an. Wir wollen sicherstellen, dass die `index`-Action unseres `index-Controller` des `HelloWorld`-Moduls das richtige `ViewModel` für das Rendering bereitstellt. Der Controller bedient sich dabei des vorher erstellten `GreetingService` um die “Greeting Line” zu erzeugen und diese über den Key `greetingLine` für das Rendering der View bereitzustellen. Wir wollen hier also 2 Dinge testen:

- Der Controller verwendet den `GreetingService` und bezieht darüber die “Greeting Line”.
- Der Controller übergibt die bezogene “Greeting Line” über den Key `greetingLine` unmodifiziert an das `ViewModel`.

Der `IndexController` mit der `indexAction`, die wir testen wollen, sieht derzeit wie folgt aus:

```
1  <?php
2
3  namespace HelloWorld\Controller;
4
5  use Zend\Mvc\Controller\AbstractActionController;
6  use Zend\View\Model\ViewModel;
7  use Zend\EventManager\EventManagerAwareInterface;
8  use HelloWorld\Service\GreetingService;
9
10 class IndexController extends AbstractActionController
11 {
12     public function indexAction()
13     {
14         $greetingLine = $this->getServiceLocator()
15             ->get('HelloWorld\Service\GreetingService')->getGreeting();
16
17         return new ViewModel(
18             array(
19                 'greetingLine' => $greetingLine
20             )
21         );
22     }
23 }
```

Listing 18.8

Wenn wir den Code betrachten, fällt auf, dass die `indexAction` über eine direkte Abhängigkeit zum `ServiceManager` (hier durch dessen Interface `ServiceLocator` charakterisiert) verfügt und über eine indirekte Abhängigkeit zum `GreetingService`, dem eigentlich Kollaborateur der `indexAction`, der über den `ServiceManager` bezogen wird. Um die Funktion der Action zu testen, kommen wir

auch hier wieder nicht drumherum, den `GreetingService` durch einen Fake zu ersetzen. Dazu haben wir in diesem Konstrukt grundsätzlich 2 Möglichkeiten: Entweder, wir sorgen im Rahmen des Tests dafür, dass der `ServiceManager` das Fake-Objekt zurückliefert und ansonsten wie gehabt den “echten” `GreetingService` oder aber wir setzen auch hier lieber wieder voll und ganz auf “Dependency Injection”:

```
1  <?php
2
3  namespace Helloworld\Controller;
4
5  use Zend\Mvc\Controller\AbstractActionController;
6  use Zend\View\Model\ViewModel;
7
8  class IndexController extends AbstractActionController
9  {
10     private $greetingService;
11
12     public function indexAction()
13     {
14         $greetingLine = $this->greetingService->getGreeting();
15
16         return new ViewModel(
17             array(
18                 'greetingLine' => $greetingLine
19             )
20         );
21     }
22
23     public function setGreetingService($greetingService)
24     {
25         $this->greetingService = $greetingService;
26     }
27
28     public function getGreetingService()
29     {
30         return $this->greetingService;
31     }
32 }
```

Listing 18.9

Jetzt müssen wir allerdings noch dafür sorgen, dass der `ControllerManager`, die Komponente des Frameworks, die sich um das Instanzieren der Controller kümmert, über die `module.config.php` des `Helloworld`-Moduls instruiert wird, sich beim `IndexController` einer Factory zu bedienen


```
1  <?php
2  // [...]
3  'controllers' => array(
4      'factories' => array(
5          'Helloworld\Controller\Index'
6              => 'Helloworld\Controller\IndexControllerFactory'
7      )
8  )
9  // [...]
```

Listing 18.10

die dem IndexController den GreetingService injiziert:

```
1  <?php
2  namespace Helloworld\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class IndexControllerFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $ctr = new IndexController();
12         $serviceLocator = $serviceLocator->getServiceLocator();
13
14         $greetingSrv = $serviceLocator
15             ->get('Helloworld\Service\GreetingService');
16
17         $ctr->setGreetingService($greetingSrv);
18         return $ctr;
19     }
20 }
```

Listing 18.11

Nun können wir dem IndexController im Rahmen des IndexControllerTest problemlos ein Fake-Objekt unterjubeln:

```
1  <?php
2  namespace HelloWorldTest\ControllerTest;
3
4  use HelloWorldTest\ControllerTest\IndexControllerTest\GreetingServiceFake;
5  use HelloWorldTest\Controller\IndexController;
6  use Zend\Http\Request;
7  use Zend\Mvc\MvcEvent;
8  use Zend\Mvc\Router\RouteMatch;
9
10 class IndexControllerTest extends \PHPUnit_Framework_TestCase
11 {
12     private $controller;
13     private $request;
14     private $routeMatch;
15     private $event;
16
17     public function setUp()
18     {
19         $this->controller = new IndexController();
20         $this->request = new Request();
21         $this->routeMatch = new RouteMatch(array('controller' => 'index'));
22         $this->event = new MvcEvent();
23         $this->event->setRouteMatch($this->routeMatch);
24         $this->controller->setEvent($this->event);
25     }
26
27     public function testIndexAction()
28     {
29         $greetingServiceFake = new GreetingServiceFake();
30         $this->controller->setGreetingService($greetingServiceFake);
31         $this->routeMatch->setParam('action', 'index');
32         $response = $this->controller->dispatch($this->request);
33         $viewModelValues = $response->getVariables();
34
35         $this->assertEquals(
36             $viewModelValues['greetingLine'],
37             'Fake Greeting Line'
38         );
39     }
40 }
```

Listing 18.12

Schauen wir uns in Ruhe an, was hier eigentlich passiert: In der `setUp`-Methode, die die notwendigen Vorbereitungen für alle `test*()`-Methoden der Testklasse vornimmt, werden zunächst einige Framework-Objekte instanziiert und initialisiert, die erforderlich sind, um die `dispatch()`-Funktion des `IndexController` aufzurufen. Dies sind im Detail und der Reihe nach:

- Der `IndexController` selbst.
- Das `Request`-Objekt.
- Das `RouteMatch`-Objekt, das sonst vom Router bereitgestellt wird, nachdem eine passende Route identifiziert wurde. Der Ordnung halber und um einen konsistenten Zustand zu gewährleisten, sorgen wir auch dafür, dass der `key controller` auf den korrekten Wert gesetzt wird, auch wenn das für die Ausführung des Tests nicht zwingend erforderlich wäre. Später setzen wir auch noch den erforderlichen `action`-Key auf `index`.
- Das `MvcEvent`-Objekt, das “Spezial-Event-Objekt” der MVC-Requestverarbeitung, das Zugriff auf die anderen wichtigen Objekte gewährt, etwa das `RouteMatch`-Objekt.

Der Test selbst besteht dann nur noch aus der Instanzierung des Controllers, der Injizierung des `GreetingServiceFake` sowie des Dispatchings, also dem Aufruf von dessen `dispatch()`-Methode, über die wir dann als Ergebnis das `ViewModel` zurückbekommen. Auch verwenden wir für diesen Test wieder ein gemeinsames Interface für den “echten” Service und dessen “Fake”:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  use HelloWorld\Service\GreetingService\HourProviderInterface;
5
6  interface GreetingServiceInterface
7  {
8      public function getGreeting();
9      public function setHourProvider(HourProviderInterface $hourProvider);
10     public function getHourProvider();
11 }
```

Listing 18.13

Der `GreetingServiceFake` sieht wie folgt aus:

```
1  <?php
2  namespace HelloWorldTest\ServiceTest\GreetingServiceTest;
3
4  use HelloWorld\Service\GreetingService\HourProviderInterface;
5  use HelloWorld\Service\GreetingServiceInterface;
6
7  class GreetingServiceFake implements GreetingServiceInterface
8  {
9      public function getGreeting()
10     {
11         return "Fake Greeting Line";
12     }
13
14     public function setHourProvider(HourProviderInterface $hourProvider)
15     {
16         return;
17     }
18
19     public function getHourProvider()
20     {
21         return;
22     }
23 }
```

Listing 18.14

Er ist so konzipiert, dass er schlichtweg einen festen String zurückgibt, auf den wir testen. Nachdem wir die `phpunit.xml` dahingehend erweitert haben, dass sie auch den neuen Test berücksichtigt

```
1  <phpunit bootstrap="./bootstrap.php">
2      <testsuites>
3          <testsuite name="AllTests">
4              <directory>./HelloworldTest/ServiceTest</directory>
5              <directory>./HelloworldTest/ControllerTest</directory>
6          </testsuite>
7      </testsuites>
8  </phpunit>
```

können wir nun das folgende Kommando im `tests`-Verzeichnis aufrufen, um auch den neuen Test auszuführen:

```
1 $ phpunit
2 > PHPUnit 3.6.12 by Sebastian Bergmann.
3 > Configuration read from /vagrant/tests/phpunit.xml
4 > ..
5 > Time: 1 second, Memory: 10.50Mb
6 > OK (2 tests, 26 assertions)
```

Der Test ist soweit also vollkommen okay und tut seinen Dienst. Allerdings gibt es ein Risiko, das wir so in unserem Test eingehen, nämlich, dass der Controller zwar mit dem String “Fake Greeting Line” arbeitet, aber diesen String gar nicht über den `GreetingService` bezogen hat. Das ist zwar jetzt vielleicht schon ein wenig abwegig, aber dennoch ein valider Punkt, denn der Test würde auch positiv verlaufen, wenn ich Folgendes tun würde:

```
1 <?php
2 // [...]
3 public function indexAction()
4 {
5     //$greetingLine = $this->greetingService->getGreeting();
6     $greetingLine = 'Fake Greeting Line';
7
8     return new ViewModel(
9         array(
10             'greetingLine' => $greetingLine
11         )
12     );
13 }
14 // [...]
```

Listing 18.15

Wenn der `GreetingService` jetzt mehr tun würde, als nur den String “Fake Greeting Line” zurückzugeben, etwa eine Transaktion anstoßen, die weitere Auswirkungen hat, würde unser Test zwar nicht fehlschlagen, das Ziel des Codes wäre aber eigentlich gar nicht erreicht.

Wenn wir also testen wollen, ob die Methode eines Objekts wirklich aufgerufen wurde, müssten wir den Zugriff protokollieren und ebenfalls überprüfen. Das können wir händisch machen, in dem wir den `GreetingServiceFake` um das entsprechende Flag erweitern:

```
1  <?php
2  namespace HelloWorldTest\ControllerTest\IndexControllerTest;
3
4  use HelloWorld\Service\GreetingService\HourProviderInterface;
5  use HelloWorld\Service\GreetingServiceInterface;
6
7  class GreetingServiceFake implements GreetingServiceInterface
8  {
9      private $getGreetingWasCalled = false;
10
11     public function getGreeting()
12     {
13         $this->getGreetingWasCalled = true;
14         return "Fake Greeting Line";
15     }
16
17     public function setHourProvider(HourProviderInterface $hourProvider)
18     {
19         return;
20     }
21
22     public function getHourProvider()
23     {
24         return;
25     }
26
27     public function setGetGreetingWasCalled($getGreetingWasCalled)
28     {
29         $this->getGreetingWasCalled = $getGreetingWasCalled;
30     }
31
32     public function getGetGreetingWasCalled()
33     {
34         return $this->getGreetingWasCalled;
35     }
36 }
```

Listing 18.16

Sowie den Test entsprechend überarbeiten:

```
1  <?php
2  // [...]
3  public function testIndexAction()
4  {
5      $greetingServiceFake = new GreetingServiceFake();
6      $this->controller->setGreetingService($greetingServiceFake);
7      $this->routeMatch->setParam('action', 'index');
8      $response = $this->controller->dispatch($this->request);
9      $viewModelValues = $response->getVariables();
10
11     $this->assertEquals(
12         $viewModelValues['greetingLine'],
13         'Fake Greeting Line'
14     );
15
16     $this->assertTrue(
17         $greetingServiceFake->getGetGreetingWasCalled()
18     );
19 }
20 // [...]
```

Listing 18.17

Die letzte Zeile ist hier neu. Sie stellt sicher, dass die entsprechende Methode auch tatsächlich aufgerufen wurde. Ist dies nicht der Fall, schlägt der Test fehl:

```
1  $ phpunit
2  > PHPUnit 3.6.12 by Sebastian Bergmann.
3  > Configuration read from /vagrant/tests/phpunit.xml
4  > .F
5  > Time: 2 seconds, Memory: 10.75Mb
6  > There was 1 failure:
7  > 1) HelloworldTest\ControllerTest\IndexControllerTest::testIndexAction
8  > Failed asserting that false is true.
```

Nur dann, wenn im Controller auch tatsächlich die Methode `getGreeting()` des `GreetingService` verwendet wird, laufen alle Tests erfolgreich durch.

Wenn wir noch einmal auf die Einleitung dieses Kapitels zurückkommen, so haben wir es in diesem Fall also mit einem “Fake”-Objekt vom Typ “Mock” zutun und nicht mehr nur mit einem “Stub”. Warum? Weil uns der `GreetingServiceFake` aktiv dabei hilft, sicherzustellen, dass der `IndexController` das tut, was er soll, nämlich den entsprechenden Service tatsächlich auch aufzurufen.

Da dies eine häufige Problemstellung bei Tests ist und die händisch Erstellung solcher Mocks doch einen gewissen Aufwand darstellt, kommt PHPUnit hier mit sehr hilfreicher Unterstützung. Es erlaubt uns, dynamisch einen Mock zu erzeugen:

```
1  <?php
2  namespace HelloworldTest\ControllerTest;
3
4  use HelloworldTest\ControllerTest\IndexControllerTest\GreetingServiceFake;
5  use HelloworldTest\Controller\IndexController;
6  use Zend\Http\Request;
7  use Zend\Mvc\MvcEvent;
8  use Zend\Mvc\Router\RouteMatch;
9
10 class IndexControllerTest extends \PHPUnit_Framework_TestCase
11 {
12     private $controller;
13     private $request;
14     private $routeMatch;
15     private $event;
16
17     public function setUp()
18     {
19         $this->controller = new IndexController();
20         $this->request = new Request();
21         $this->routeMatch = new RouteMatch(array('controller' => 'index'));
22         $this->event = new MvcEvent();
23         $this->event->setRouteMatch($this->routeMatch);
24         $this->controller->setEvent($this->event);
25     }
26
27     public function testIndexAction()
28     {
29         $greetingServiceFake = $this->getMock(
30             'GreetingServiceFake', array('getGreeting')
31         );
32
33         $greetingServiceFake->expects($this->once())
34             ->method('getGreeting')
35             ->will($this->returnValue('Fake Greeting Line'));
36
37         $this->controller->setGreetingService($greetingServiceFake);
38         $this->routeMatch->setParam('action', 'index');
39         $response = $this->controller->dispatch($this->request);
```



```
40
41         $viewModelValues = $response->getVariables();
42
43         $this->assertEquals(
44             $viewModelValues['greetingLine'],
45             'Fake Greeting Line'
46         );
47     }
48 }
```

Listing 18.18

Auf diese Weise brauchen wir unseren `GreetingServiceFake` im Grunde schon gar nicht mehr, das Testen wird noch einfacher und die Hemmschwelle, den Test der Bequemlichkeit halber auszulassen, weil er ja zusätzliche Aufwände erzeugt, sinkt. Wenn der `GreetingService` jetzt planmäßig genutzt wird sieht die Ausgabe wie folgt aus:

```
1 $ phpunit
2 > PHPUnit 3.6.12 by Sebastian Bergmann.
3 > Configuration read from /vagrant/tests/phpunit.xml
4 > ..
5 > Time: 1 second, Memory: 11.00Mb
6 > OK (2 tests, 27 assertions)
```

Andernfalls schlägt PHPUnit auch hier wieder Alarm:

```
1 $ phpunit
2 > PHPUnit 3.6.12 by Sebastian Bergmann.
3 > Configuration read from /vagrant/tests/phpunit.xml
4 > .F
5 > Time: 0 seconds, Memory: 11.00Mb
6 > There was 1 failure:
7 > 1) HelloworldTest\ControllerTest\IndexControllerTest::testIndexAction
8 > Expectation failed for method name is equal to
9 > <string:getGreeting> when invoked 1 time(s).
10 > Method was expected to be called 1 times, actually called 0 times.
11 > FAILURES!
12 > Tests: 2, Assertions: 27, Failures: 1.
```

Und selbstverständlich können wir so auch den `HourProviderFake` von oben obsolet machen und ihn stattdessen dynamisch von PHPUnit erzeugen lassen.

19 Benutzerverwaltung

19.1 Einleitung

Die meisten Anwendungen sind irgendwie interaktiv. Wenn sie es nicht wären, dann wäre das World Wide Web wohl auch nicht das, was es heute ist. Interaktion erfordert fast immer, dass klar ist, wer da eigentlich mit wem interagiert und wer was im System darf. Und darum geht es in diesem Kapitel.

19.2 Nutzer-Authentifizierung

Der Prozess der Authentifizierung ist der Nachweis, dass eine Person auch die ist, für die sie sich ausgibt. Ein Nutzer, der sich in einen geschützten Bereich einloggt (etwa sowas wie "Mein eBay" oder ähnliches) gibt dazu in aller Regel zunächst einen identifizierenden Bezeichner ein, z.B. seinen Nutzernamen oder E-Mail-Adresse (und gibt damit vor, eine bestimmte Person zu sein), sowie den dazu passenden, geheimen "Schlüssel", den nur er, sonst aber Niemand kennt - sein Passwort also. Die Tatsache, dass die Person, die sich versucht einzuloggen, über das Wissen des passenden, geheimen Schlüssels verfügt, reicht aus, um ihn als denjenigen zu bestätigen, den er angibt zu sein. Um diese Überprüfung vornehmen zu können, benötigt das System, an dem sich der Nutzer anmeldet, die entsprechenden Vergleichswerte.

Das Zend Framework 2 bringt dazu `Zend\Authentication` mit, wobei dessen `AuthenticationService` die zentrale Instanz darstellt, der die Aufgabe zuteil wird, festzustellen, ob eine Person auch wirklich die ist, die sie zu sein scheint. Dafür bedient sich der Service wiederum eines "Adapters", der die eingegebenen Daten entgegennimmt und mit den passenden Vergleichswerten aus der Datenquelle der Wahl abgleicht. Häufig kommt dabei eine Datenbank zum Einsatz, allerdings liefert das Framework auch Adapter für die HTTP-basierte Authentifizierung oder die über ein externes LDAP-System mit. Das `AdapterInterface` kann für die eigene Implementierung andersartiger Datenquellen herangezogen werden.

Wenn wir unser `HelloWorld`-Modul um eine Nutzer-Authentifizierung erweitern wollen, erstellen wir dazu zunächst den notwendigen Controller samt Action mit rudimentärer Verarbeitungslogik

```
1  <?php
2  namespace HelloWorld\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6
7  class AuthController extends AbstractActionController
8  {
9      private $loginForm;
10
11     public function loginAction()
12     {
13         if (!$this->loginForm)
14             throw new \BadMethodCallException('Login Form not yet set!');
15
16         if ($this->getRequest()->isPost()) {
17             $this->loginForm->setData($this->getRequest()->getPost());
18
19             if ($this->loginForm->isValid()) {
20                 var_dump($this->loginForm->getData());exit;
21             } else {
22                 return new ViewModel(
23                     array(
24                         'form' => $this->loginForm
25                     )
26                 );
27             }
28         } else {
29             return new ViewModel(
30                 array(
31                     'form' => $this->loginForm
32                 )
33             );
34         }
35     }
36
37     public function setLoginForm($loginForm)
38     {
39         $this->loginForm = $loginForm;
40     }
41
42     public function getLoginForm()
```

```
43     {
44         return $this->loginForm;
45     }
46 }
```

Listing 19.1

sowie die dazu passende Route in der `module.config.php`:

```
1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'login' => array(
6              'type' => 'Literal',
7              'options' => array(
8                  'route' => '/login',
9                  'defaults' => array(
10                     'controller' => 'Helloworld\Controller\Auth',
11                     'action' => 'login',
12                 ),
13             ),
14         ),
15     ),
16 ),
17 // [...]
```

Listing 19.2

Nicht vergessen, auch das passende View-Script anzulegen, wenn auch zunächst erst einmal ohne Inhalt. Fehlt es, kommt es zu einem Fehler.

Jetzt benötigen wir das eigentliche Login-Formular

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Form;
5
6  class Login extends Form
7  {
8      public function __construct()
9      {
```

```
10     parent::__construct('login');
11     $this->setAttribute('action', '/login');
12     $this->setAttribute('method', 'post');
13     $this->setInputFilter(new \Helloworld\Form\LoginFilter());
14
15     $this->add(array(
16         'name' => 'username',
17         'attributes' => array(
18             'type' => 'text',
19         ),
20         'options' => array(
21             'label' => 'Benutzername:',
22         )
23     ));
24
25     $this->add(array(
26         'name' => 'password',
27         'attributes' => array(
28             'type' => 'password',
29         ),
30         'options' => array(
31             'label' => 'Password:',
32         ),
33     ));
34
35     $this->add(array(
36         'name' => 'submit',
37         'attributes' => array(
38             'type' => 'submit',
39             'value' => 'Einloggen'
40         ),
41     ));
42 }
43 }
```

Listing 19.3

sowie den passenden “InputFilter”:

```
1  <?php
2  namespace Helloworld\Form;
3
4  use Zend\Form\Form;
5  use Zend\InputFilter\InputFilter;
6
7  class LoginFilter extends InputFilter
8  {
9      public function __construct()
10     {
11         $this->add(array(
12             'name' => 'username',
13             'required' => true,
14         ));
15
16         $this->add(array(
17             'name' => 'password',
18             'required' => true,
19         ));
20     }
21 }
```

Listing 19.4

Das Formular erwartet die Eingabe des Nutzernamens und des Passworts, wobei beide Angaben verpflichtend sind. In der login-View sorgen wir für die Darstellung des Formulars:

```
1  <?php
2  $this->form->prepare();
3  echo $this->form()->openTag($this->form);
4  echo $this->formRow($this->form->get('username'));
5  echo $this->formRow($this->form->get('password'));
6  echo $this->formSubmit($this->form->get('submit'));
7  echo $this->form()->closeTag();
```

Listing 19.5

Die Form injizieren wir über eine Factory in den Controller:

```
1  <?php
2  namespace HelloWorld\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class AuthControllerFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $ctr = new AuthController();
12         $ctr->setLoginForm(new \HelloWorld\Form\Login());
13         return $ctr;
14     }
15 }
```

Listing 19.6

Rufen wir nun die URL /login auf, so sehen wir bereits das Login-Formular und die Daten werden nach Absenden des Formulars auch bereits korrekt übertragen.

Jetzt gilt es, die eingegebenen Daten zu überprüfen. Dazu bereiten wir den AuthenticationService über eine eigene Factory für seinen Einsatz vor:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6  use Zend\Authentication\Storage\NonPersistent;
7  use Zend\Authentication\Adapter\DbTable;
8
9  class AuthServiceFactory implements FactoryInterface
10 {
11     const TABLE_NAME = "users";
12     const IDENTITY_COLUMN = "username";
13     const CREDENTIAL_COLUMN = "password";
14
15     public function createService(ServiceLocatorInterface $serviceLocator)
16     {
17         $dbAdapter = $serviceLocator->get('Zend\Db\Adapter\Adapter');
18
19         $service = new \Zend\Authentication\AuthenticationService(
```

```

20         new NonPersistent(),
21         new DbTable(
22             $dbAdapter,
23             self::TABLE_NAME,
24             self::IDENTITY_COLUMN,
25             self::CREDENTIAL_COLUMN,
26         )
27     );
28
29     return $service;
30 }
31 }

```

Listing 19.7

Die Factory wiederum setzt einen fertig konfigurierten DB-Adapter voraus, den ich ebenfalls in der `module.config.php` eingerichtet habe:

```

1  <?php
2  // [...]
3  'service_manager' => array(
4      'factories' => array(
5          'Zend\Db\Adapter\Adapter' => function ($sm) {
6              $config = $sm->get('Config');
7              $dbParams = $config['dbParams'];
8
9              return new Zend\Db\Adapter\Adapter(array(
10                 'driver'      => 'pdo',
11                 'dsn'         =>
12                     'mysql:dbname=' . $dbParams['database'] .
13                     ';host=' . $dbParams['hostname'],
14                 'database'   => $dbParams['database'],
15                 'username'   => $dbParams['username'],
16                 'password'   => $dbParams['password'],
17                 'hostname'   => $dbParams['hostname'],
18             ));
19         },
20     )
21 )

```

Listing 19.8

Die eigentlichen Verbindungsdaten werden dabei aus einer externen Konfigurationsdatei geladen. Wenn wir den AuthController noch um die Eigenschaft \$authService samt Getter/Setter-Methoden erweitern, können wir auch den AuthService über die AuthControllerFactory injizieren:

```
1  <?php
2  namespace HelloWorld\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class AuthControllerFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $ctr = new AuthController();
12         $ctr->setLoginForm(new \HelloWorld\Form\Login());
13
14         $ctr->setAuthService($serviceLocator
15                             ->getServiceLocator()
16                             ->get('HelloWorld\Service\AuthService'));
17
18         return $ctr;
19     }
20 }
```

Listing 19.9

Der HelloWorld\Service\AuthService wurde dabei zuvor über die module.config.php beim ServiceManager registriert. Hier versteckt sich übrigens eine Falle, die einen schon einmal zum Haareraufen bringen kann: Der \$serviceLocator, der an die AuthControllerFactory übergeben wird ist, ist nicht, wie sonst bei “Service-Factories” üblich, der ServiceManager, sondern der ControllerManager. Daher würde ein \$serviceLocator->get() hier fehlschlagen. Es muss zuerst vom ControllerManager auf den ServiceManager zugegriffen und dort dann der eigentliche Service angefordert werden.

Wenn wir nun noch eine passende Datenstruktur erstellen

```
1 CREATE TABLE users (  
2     id int(10) NOT NULL auto_increment,  
3     username varchar(30) NOT NULL,  
4     password varchar(50) NOT NULL, PRIMARY KEY (id)  
5 );
```

und mit einigen Testdaten versehen

```
1 INSERT INTO users (username, password)  
2 VALUES ("testuser", "testpassword");
```

können wir uns bereits einloggen:

```
1 object(Zend\Authentication\Result)#253 (3) {  
2     ["code":protected]=> int(1)  
3     ["identity":protected]=> string(4) "testuser"  
4     ["messages":protected]=> array(1) {  
5         [0]=> string(26) "Authentication successful."  
6     }  
7 }
```

Bei fehlerhaften Eingaben sehen wir die entsprechende Fehlermeldung:

```
1 object(Zend\Authentication\Result)#253 (3) {  
2     ["code":protected]=> int(-3)  
3     ["identity":protected]=> string(8) "testuser"  
4     ["messages":protected]=> array(1) {  
5         [0]=> string(31) "Supplied credential is invalid."  
6     }  
7 }
```

Soweit so okay, aber noch nicht wirklich gut. Für den Fall, dass ungültige Daten eingegeben wurden, wollen wir wieder das Login-Formular samt Fehlermeldung anzeigen. Im Erfolgsfall eine Begrüßungszeile, aber nicht mehr das Formular. Wir erweitern also den AuthController

```
1  <?php
2
3  namespace HelloWorld\Controller;
4
5  use Zend\Mvc\Controller\AbstractActionController;
6  use Zend\View\Model\ViewModel;
7
8  class AuthController extends AbstractActionController
9  {
10     private $loginForm;
11     private $authService;
12
13     public function loginAction()
14     {
15         if (!$this->loginForm)
16             throw new \BadMethodCallException('Login Form not yet set!');
17         if (!$this->authService)
18             throw new \BadMethodCallException('Auth Service not yet set!');
19
20         if ($this->getRequest()->isPost()) {
21             $this->loginForm->setData($this->getRequest()->getPost());
22
23             if ($this->loginForm->isValid()) {
24                 $data = $this->loginForm->getData();
25
26                 $this->authService->getAdapter()
27                     ->setIdentity($data['username']);
28
29                 $this->authService->getAdapter()
30                     ->setCredential($data['password']);
31
32                 $authResult = $this->authService->authenticate();
33
34                 if (!$authResult->isValid())
35                 {
36                     return new ViewModel(
37                         array(
38                             'form' => $this->loginForm,
39                             'loginError' => true
40                         )
41                     );
42                 }
43             }
44         }
45     }
46 }
```

```
43         else
44             return new ViewModel(
45                 array(
46                     'loginSuccess' => true,
47                     'userLoggedIn'
48                         => $authResult->getIdentity()
49                 )
50             );
51
52     } else {
53         return new ViewModel(
54             array(
55                 'form' => $this->loginForm
56             )
57         );
58     }
59 } else {
60     return new ViewModel(
61         array(
62             'form' => $this->loginForm
63         )
64     );
65 }
66
67
68 public function setLoginForm($loginForm)
69 {
70     $this->loginForm = $loginForm;
71 }
72
73 public function getLoginForm()
74 {
75     return $this->loginForm;
76 }
77
78 public function setAuthService($authService)
79 {
80     $this->authService = $authService;
81 }
82
83 public function getAuthService()
84 {
```

```
85         return $this->authService;
86     }
87 }
```

Listing 19.10

sowie das View-File:

```
1  <?php
2  if ($this->loginSuccess)
3      echo "Hallo, " . $this->userLoggedIn . '!';
4  else
5  {
6      if ($this->loginError)
7          echo "Die eingegebenen Daten sind ungültig.";
8
9      $this->form->prepare();
10     echo $this->form()->openTag($this->form);
11     echo $this->formRow($this->form->get('username'));
12     echo $this->formRow($this->form->get('password'));
13     echo $this->formSubmit($this->form->get('submit'));
14     echo $this->form()->closeTag();
15 }
```

Listing 19.11

Derzeit speichern wir das Passwort noch im Klartext in der Datenbank. Das ist irgendwas zwischen unklug und grob fahrlässig und sollten wir tunlichst vermeiden. Auch wenn es sicherlich nicht optimal ist, beschränken wir uns für das folgende Beispiel auf das Hashing von Passwörtern auf Basis des MD5-Algorithmus. Dazu fügen wir die Testdaten entsprechend mit verschlüsseltem Passwort ein

```
1  INSERT INTO users (username, password)
2  VALUES ("testuser", md5("testpassword"));
```

und passen den DbTable-Adapter dahingehend an, das er md5(?) als “Credential Treatment” verwendet:

```
1  <?php
2  namespace HelloWorld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6  use Zend\Authentication\Storage\NonPersistent;
7  use Zend\Authentication\Adapter\DbTable;
8
9  class AuthServiceFactory implements FactoryInterface
10 {
11     const TABLE_NAME = "users";
12     const IDENTITY_COLUMN = "username";
13     const CREDENTIAL_COLUMN = "password";
14
15     public function createService(ServiceLocatorInterface $serviceLocator)
16     {
17         $dbAdapter = $serviceLocator->get('Zend\Db\Adapter\Adapter');
18
19         $service = new \Zend\Authentication\AuthenticationService(
20             new NonPersistent(),
21             new DbTable($dbAdapter,
22                 self::TABLE_NAME,
23                 self::IDENTITY_COLUMN,
24                 self::CREDENTIAL_COLUMN,
25                 'md5(?)'
26             )
27         );
28
29         return $service;
30     }
31 }
```

Listing 19.12

Neu ist hier das 'md5(?)'. Das Fragezeichen wird durch den Adapter dann dynamisch durch das angegebene "Credential" ersetzt, wenn das entsprechende SQL-Statement erzeugt wird.

19.3 Nutzer-Sessions

In aller Regel führt eine erfolgreiche Nutzer-Authentifizierung zu einer Nutzer-Session, also einem Zeitraum, in dem jede Interaktion mit der Anwendung in Namen des zuvor einmalig authentifizierten Nutzers geschieht. Ich muss also nicht für jede Aktion erneut Nutzernamen und Passwort

eingeben, sondern eben nur einmalig und das für einen Zeitraum von Minuten bis hin zu einigen Stunden oder Tagen, je nach Konfiguration.

Der Code oben, der für die Authentifizierung des Nutzers sorgt, erzeugt derzeit noch keine Session, weil im Zuge der Erzeugung des AuthenticationService die NonPersistent-Implementierung als “Storage Engine” genutzt wird:

```

1  <?php
2  $service = new \Zend\Authentication\AuthenticationService(
3      new NonPersistent(),
4      new DbTable($dbAdapter,
5          self::TABLE_NAME,
6          self::IDENTITY_COLUMN,
7          self::CREDENTIAL_COLUMN,
8          'md5(?)'
9      )
10 );

```

Listing 19.13

Der AuthenticationService verwendet ein sog. “Storage”-Objekt, um das (positive) Ergebnis einer Authentifizierung für einen längeren Zeitraum vorzuhalten. Üblicherweise kommt als “Storage” eine PHP-Session zum Einsatz, in der der Identifikator, also etwa der Nutzernamen des eingeloggten Nutzers, gespeichert wird. Dazu muss die AuthServiceFactory von oben wie folgt erweitert werden:

```

1  <?php
2  namespace HelloWorld\Service;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6  use Zend\Authentication\Storage\Session;
7  use Zend\Authentication\Adapter\DbTable;
8
9  class AuthServiceFactory implements FactoryInterface
10 {
11     const TABLE_NAME = "users";
12     const IDENTITY_COLUMN = "username";
13     const CREDENTIAL_COLUMN = "password";
14
15     public function createService(ServiceLocatorInterface $serviceLocator)
16     {
17         $dbAdapter = $serviceLocator->get('Zend\Db\Adapter\Adapter');
18

```

```

19         $service = new \Zend\Authentication\AuthenticationService(
20             new Session(),
21             new DbTable($dbAdapter,
22                 self::TABLE_NAME,
23                 self::IDENTITY_COLUMN,
24                 self::CREDENTIAL_COLUMN,
25                 'md5(?)'
26             )
27         );
28
29         return $service;
30     }
31 }

```

Listing 19.14

Anstelle von NonPersistent kommt jetzt Session-Storage zum Einsatz, was zur Folge hat, dass nun nach einem erfolgreichen Login eine Session auf dem Server erzeugt wird und der Browser des Nutzers das passende Cookie enthält. Jetzt können wir etwa dafür sorgen, dass ein Nutzer, der die URL /login aufruft, sich aber zuvor bereits erfolgreich eingeloggt hat, nicht mehr das Formular, sondern eine Begrüßung sieht:

```

1  <?php
2
3  namespace Helloworld\Controller;
4
5  use Zend\Mvc\Controller\AbstractActionController;
6  use Zend\View\Model\ViewModel;
7
8  class AuthController extends AbstractActionController
9  {
10     private $loginForm;
11     private $authService;
12
13     public function loginAction()
14     {
15         if ($this->authService->hasIdentity())
16         {
17             return new ViewModel(
18                 array(
19                     'loginSuccess' => true,
20                     'userLoggedIn'
21                     => $this->authService->getIdentity()

```



```
22         )
23     );
24 }
25
26 if (!$this->loginForm)
27     throw new \BadMethodCallException('Login Form not yet set!');
28
29 if (!$this->authService)
30     throw new \BadMethodCallException('Auth Service not yet set!');
31
32 if ($this->getRequest()->isPost()) {
33     $this->loginForm->setData($this->getRequest()->getPost());
34
35     if ($this->loginForm->isValid()) {
36
37         $data = $this->loginForm->getData();
38
39         $this->authService->getAdapter()
40             ->setIdentity($data['username']);
41
42         $this->authService->getAdapter()
43             ->setCredential($data['password']);
44
45         $authResult = $this->authService->authenticate();
46
47         if (!$authResult->isValid())
48         {
49             return new ViewModel(
50                 array(
51                     'form' => $this->loginForm,
52                     'loginError' => true
53                 )
54             );
55         }
56     else
57         return new ViewModel(
58             array(
59                 'loginSuccess' => true,
60                 'userLoggedIn'
61                     => $authResult->getIdentity()
62             )
63         );
```

```

64
65         } else {
66             return new ViewModel(
67                 array(
68                     'form' => $this->loginForm
69                 )
70             );
71         }
72     } else {
73         return new ViewModel(
74             array(
75                 'form' => $this->loginForm
76             )
77         );
78     }
79 }
80 // [...]
81 }

```

Listing 19.15

Was jetzt eigentlich nur noch fehlt, ist eine “Logout”-Funktion, die sich sehr einfach als weitere Action im AuthController realisieren lässt:

```

1  <?php
2  // [...]
3  public function logoutAction()
4  {
5      if ($this->authService->hasIdentity())
6          $this->authService->clearIdentity();
7
8      $this->redirect()->toUrl('/login');
9  }
10 // [...]

```

Listing 19.16

Zugegeben, die Implementierung der loginAction() ist mittlerweile nicht mehr sonderlich gut verständlich. Findest du einen Weg, um ihn weiter zu verbessern?

19.4 Ressourcen & Rollen

Meistens soll nicht jeder Nutzer in einem System alles können. Ein Administrator etwa hat in aller Regel mehr Funktionen zur Verfügung, als der “normale” Nutzer. Um diese Beschränkungen

abzubilden, empfiehlt sich der Einsatz von `Zend\Permissions\Acl`. Darüber lassen sich zum einen Rollen definieren, die ein eingeloggter Nutzer einnimmt, sowie die sog. “Ressourcen”, auf die zugegriffen werden soll. Auf Basis dieser Informationen lassen sich dann logische Verbindungen herstellen, also welcher Nutzer welche Rolle einnimmt und welcher Rolle wiederum der Zugriff auf welche Ressourcen gewährt wird.

Nehmen wir an, wir benötigen die Möglichkeit, eine Liste aller im System verfügbaren Nutzer anzuzeigen. Diese Liste soll aber nur Nutzern zur Verfügung stehen, die die Rolle des Administrators einnehmen. Für alle anderen Nutzer soll die Liste nicht verfügbar sein. Wir könnten dazu den entsprechenden Regelsatz aufbauen. Zunächst definieren wir die beiden Nutzerrollen im System:

```
1 <?php
2 $acl = new \Zend\Permissions\Acl\Acl;
3 $guestRole = new \Zend\Permissions\Acl\Role\GenericRole('guest');
4 $adminRole = new \Zend\Permissions\Acl\Role\GenericRole('admin');
5 $acl->addRole($guestRole);
6 $acl->addRole($adminRole, $guestRole);
```

Listing 19.17

Die “Admin-Rolle” ist hier so angelegt, dass sie die Rolle des “guest” konzeptionell erweitert, also etwaige Rechte des “guest” automatisch vererbt bekommt. Nun erstellen wir noch die Ressource:

```
1 <?php
2 // [...]
3 $usersPage = new
4     \Zend\Permissions\Acl\Resource\GenericResource('usersPage');
5
6 $acl->addResource($usersPage);
```

Listing 19.18

und definieren, welche Rolle auf die Ressource zugreifen darf:

```
1 <?php
2 // [...]
3 $acl->allow($adminRole, $usersPage, 'view');
```

Listing 19.19

Wobei `view` hier die sog. “Privilege” darstellt, als die Aktion, die mit der jeweiligen Ressource ausgeführt werden darf. Hier empfiehlt es sich häufig, die üblich CRUD-Privilegien zu modellieren, es können aber, wie bei den Ressourcen oder Rollen auch, freie Bezeichner gewählt werden. Mehrere Privilegien lassen sich in einem Rutsch über Verwendung eines Arrays definieren.

Über die `isAllowed()`-Methode der erzeugten `Acl` lassen sich dann entsprechende Abfragen auf die Verfügbarkeit einer Ressource realisieren:

```

1  <?php
2  // [...]
3  if(!$acl->isAllowed('admin', 'usersPage', 'view'))
4      throw new \DomainException('Resource not available');

```

Listing 19.20

Jetzt stellt sich im Grunde nur noch die Frage, an welchen Stellen in der Anwendung man am besten die Konfiguration von oben ablegt und wie die jeweils notwendigen Vergleichsdaten vorgehalten werden, bzw. aus welchen Informationen man sie dynamisch ermittelt. Hier fällt die Antwort sehr juristisch aus: Es kommt drauf an. Es kommt drauf an, wie man den Code strukturiert, ob es etwa praktikabel ist, ganze Controller als eine Ressource zu definieren, oder ob es “feingranular” auf Basis von Actions passieren muss. Auch Services oder etwa einzelne Entities können als Ressourcen modelliert werden. Manchmal ist sogar das andere Extrem denkbar, nämlich ganze Module als eine Ressource zu verstehen. In unserem konkreten Beispiel von oben, in dem wir erst einmal sämtliche Befehle in der Action implementiert haben

```

1  <?php
2  // [...]
3  public function usersAction()
4  {
5      $acl = new \Zend\Permissions\Acl\Acl;
6      $guestRole = new \Zend\Permissions\Acl\Role\GenericRole('guest');
7      $adminRole = new \Zend\Permissions\Acl\Role\GenericRole('admin');
8      $acl->addRole($guestRole);
9      $acl->addRole($adminRole, $guestRole);
10
11      $usersPage =
12          new \Zend\Permissions\Acl\Resource\GenericResource('usersPage');
13
14      $acl->addResource($usersPage);
15      $acl->allow($adminRole, $usersPage, 'view');
16
17      if(!$acl->isAllowed('admin', 'usersPage', 'view'))
18          throw new \DomainException('Resource not available');
19
20      // $users = [...] Alle Nutzer aus der Datenbank laden
21
22      return new ViewModel(
23          array(
24              'users' => $users
25          )
26      );
27  }

```

Listing 19.21

würde sich in meinen Augen folgendes anbieten: Den Aufbau der ACL samt Rollen und Ressourcen verschieben die in den `service`-Abschnitt der `module.config.php` und die `AuthControllerFactory` dahingehend erweitern, dass sie die `Acl` in den `AuthController` injiziert. Die Datenbanktabelle “users” erweitern wir um das Feld “role”, in dem jeweils die Rolle des Nutzer vorgehalten wird. Soll ein Nutzer mehrere Rollen einnehmen, würde man das entsprechend über eine m:n-Beziehung mit mehreren Datenbanktabellen lösen können.

Wie im Folgenden schematisch dargestellt, könnten wir uns dann in der eigentlichen Action um das Wesentliche kümmern:

```

1  <?php
2  // [...]
3  public function usersAction()
4  {
5      if ($this->authService->hasIdentity())
6          $this->redirect()->toUrl('/login');
7
8      // $user = [...] Eingeloggten Nutzer laden
9
10     if(!$this->acl->isAllowed($user->getRole(), 'usersPage', 'view'))
11         throw new \DomainException('Resource not available');
12
13     // $users = [...] Alle Nutzer zur Anzeige aus der Datenbank laden
14
15     return new ViewModel(
16         array(
17             'users' => $users
18         )
19     );
20 }
```

Listing 19.22

Übrigens: Bislang haben wir mit `GenericRole` und `GenericResource` sozusagen “blutleere” Klassen in der `Acl` eingesetzt, die nur als Datencontainer für die `roleId` resp. `resourceId` dienen. Tatsächlich können auch “echte” Klassen, also Controller, Services, Entities & Co. die Interfaces `RoleInterface` bzw. `ResourceInterface` implementieren (sie müssen lediglich über eine öffentliche Methode ihren Bezeichner preisgeben) und so zu Ressourcen oder Rollen in der `Acl` werden. Häufig ist dieser Ansatz weniger “konstruiert” und sorgt dafür, dass keine zusätzlichen, häufig dann inkonsistent gewählten Bezeichner, in das System Einzug halten.

19.5 ZfcUser

Auch wenn `Zend\Authentication` und `Zend\Permissions\Acl` einem bereits viel Arbeit abnehmen, muss man für eine integrierte und einsetzbare Lösung wie zuvor besprochen noch eine ganze Menge weitere Handgriffe erledigen: Ein Formular zur Abfrage der Zugangsdaten sowie ein Registrierungs-Formular erstellen, das Handling fehlerhafter Eingaben realisieren oder auch eine "Passwort vergessen"-Funktion implementieren. Um nur mal exemplarisch einige Bausteine zu nennen. Weil es sich hierbei im Grunde auch immer wieder um die gleichen Aufgaben handelt, bietet es sich an, das Problem einmal zu lösen und das Ergebnis in Form eines ZF2-Moduls bereitzustellen. Das [ZF-Commons-Team](https://github.com/ZF-Commons)¹ hat sich diese Mühe mit `ZfcUser`² gemacht, das als zusätzliches Modul in eine bestehende Anwendung integriert werden kann.

Die Installation kann wieder einfach über Composer stattfinden. Dazu die `composer.json` entsprechend um die Abhängigkeiten

```
1 "zf-commons/zfc-base": "dev-master"
2 "zf-commons/zfc-user": "dev-master"
```

erweitern und

```
1 $ php composer.phar update
```

ausführen. Nicht vergessen, die beiden zusätzlichen Module über die `application.config` zu aktivieren

```
1 <?php
2 return array(
3     'modules' => array(
4         'Application',
5         'Helloworld',
6         'ZfcBase',
7         'ZfcUser'
8     ),
9     // [...]
```

Listing 19.23

und via

¹<https://github.com/ZF-Commons>

²<https://github.com/ZF-Commons/ZfcUser>

```
1 CREATE TABLE user
2 (
3     user_id      INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL,
4     username     VARCHAR(255) DEFAULT NULL UNIQUE,
5     email        VARCHAR(255) DEFAULT NULL UNIQUE,
6     display_name VARCHAR(50)  DEFAULT NULL,
7     password     VARCHAR(128) NOT NULL
8 ) ENGINE=InnoDB;
```

die notwendige Datenstruktur zu erzeugen. Das Modul setzt zudem eine fertig konfigurierte Datenbankbindung voraus, wie wir sie weiter oben bereits erstellt haben.

Öffnet man nun die URL `/user`, so bekommt man eine fertige Lösung für das Benutzermanagement präsentiert, samt Registrierungsformular und allem, was man sonst noch so benötigt. Ein genauerer Blick in das Modul und dessen (englischsprachige) [Dokumentation](https://github.com/ZF-Commons/ZfcUser)³, lohnt sich in jeden Fall. Im [ZfcUser-Wiki](https://github.com/ZF-Commons/ZfcUser/wiki)⁴ finden sich weitere Hinweise, unter anderem auch zur Anpassung der mitgelieferten Formulare.

³<https://github.com/ZF-Commons/ZfcUser>

⁴<https://github.com/ZF-Commons/ZfcUser/wiki>

20 ZF2 auf der Kommandozeile

20.1 Einleitung

Bislang haben wir uns das Framework nur im Kontext eines HTTP-Requests angesehen. Alle Funktionalität war bislang also dann verfügbar, wenn über einen entsprechenden HTTP-Client, in aller Regel einen Webbrowser, ein Request durch den Nutzer an das Framework gesendet wurde. Was aber, wenn man bestimmte Funktionen implementieren will, für die es keinen HTTP-Request benötigt und die sich von der Kommandozeile aufrufen lassen sollen? Funktionen, die etwa im Rahmen eines Cron-Jobs regelmäßig ausgeführt werden sollen?

Das Gute ist, dass `Zend\Application` auch bereits die Nutzung über die Kommandozeile unterstützt. Und das ohne großes Zutun. In Zusammenarbeit mit `Zend\Console`, das noch einige Hilfestellung im Umgang mit der Kommandozeile mitbringt, ist es möglich, Funktionen aufzurufen, die man zuvor mit den üblichen Mitteln, also Routen, Controllern und Actions bereitgestellt hat. Das ist sehr praktisch, muss man sich doch so nicht noch an weitere APIs und spezielle Mechanismen gewöhnen.

20.2 Aufruf über die Konsole

Wenn man die `index.php`, die ja auch vom Apache (bzw. dem Webserver der Wahl) als Einstiegspunkt in die HTTP-Request-Verarbeitung durch die Applikation verwendet wird, über die Kommandozeile aufruft, sieht man zunächst das sog. "Application Banner", das so oder so ähnlich aussieht:

```
1 $ cd public
2 $ php index.php
3 > Zend Framework 2.0.3 application.
```

Wichtig ist natürlich zunächst, dass PHP auf der Kommandozeile überhaupt (am besten an beliebiger Stelle im Verzeichnissystem) verfügbar ist. Der Aufruf von

```
1 $ php --version
2 > PHP 5.3.10-1ubuntu3.2 with Suhosin-Patch (cli)
3 > (built: Jun 13 2012 17:19:58)
4 > Copyright (c) 1997-2012 The PHP Group
5 > Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies
```


gibt Aufschluss darüber, ob und wenn ja, in welcher Version PHP auf dem System verfügbar ist. Notfalls muss man hier zunächst Hand anlegen.

Auch kann jedes Modul der Anwendung ein eigenes Banner erzeugen. Wenn wir der `Module.php` die Methode `getConsoleBanner($console)` hinzufügen und dort das Interface `ConsoleBannerProviderInterface` implementieren, wird der Wert, den die Methode zurückliefert, auf der Kommandozeile ausgegeben:

```

1  <?php
2  namespace Helloworld;
3
4  use Zend\ModuleManager\ModuleManager;
5  use Zend\ModuleManager\ModuleEvent;
6  use Zend\Console\Adapter\AdapterInterface as Console;
7  use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
8
9  class Module implements ConsoleBannerProviderInterface
10 {
11     public function getConsoleBanner(Console $console)
12     {
13         return
14             "=====\n" .
15             "  Helloworld-Modul, Version 1.0  \n" .
16             "=====\\n";
17     }
18
19     // [...]
20 }
```

Listing 20.1

Noch hilfreicher als ein “Banner” sind Informationen zur Verwendung der Kommandozeilen-Funktionen des Moduls. Dazu können wir die Methode `getConsoleUsage($console)` implementieren und das `ConsoleUsageProviderInterface`-Interface implementieren. Übrigens reicht es hier - im Gegensatz zu manch’ anderer Stelle im Framework - nicht aus, wenn man lediglich die Methode implementiert. Es muss zwingend auch das entsprechende Interface über `implements` berücksichtigt werden. Exemplarisch wollen wir dafür sorgen, dass unser `Helloworld`-Modul eine ähnliche Ausgabe wie das des Shell-Kommandos “date” erzeugt (hier beispielhaft auf einer Linux-Shell ausgeführt):

```

1  $ date
2  > Fri Aug 31 14:45:09 UTC 2012
```

Die Ausgabe soll erfolgen, wenn auf der Shell das Kommando

```
1 $ php index.php show date
```

aufgerufen wird. Damit der Anwender weiß, wie er der Anwendung die Date-Ausgabe entlocken kann, legen wir die passende “Usage Information” in der `Module.php` an

```
1  <?php
2  namespace Helloworld;
3
4  use Zend\ModuleManager\ModuleManager;
5  use Zend\ModuleManager\ModuleEvent;
6  use Zend\Console\Adapter\AdapterInterface as Console;
7  use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
8
9  class Module implements ConsoleBannerProviderInterface
10 {
11     public function getConsoleBanner(Console $console)
12     {
13         return
14             "=====\n" .
15             "    Helloworld-Modul, Version 1.0    \n" .
16             "=====\n";
17     }
18
19     public function getConsoleUsage(Console $console)
20     {
21         return array(
22             'show date [--format=]'
23             => 'Zeigt das aktuelle Datum/Uhrzeit an.',
24         );
25     }
26
27     // [...]
28 }
```

Listing 20.2

und erhalten die folgende Ausgabe:

```

1 $ php index.php
2 > =====
3 >      Helloworld-Modul, Version 1.0
4 > =====
5
6 > index.php show date    Zeigt das aktuelle Datum/Uhrzeit an.
```

Noch eine kleine Erweiterung: Wir wollen es dem Anwender optional möglich machen, das angezeigte Datum in seinem Format zu beeinflussen. Weil wir für die Implementierung des Features später auf die `date()`-Funktion von PHP zurückgreifen werden, kann das Format über ein Muster definiert werden, wie es die [date\(\)-Funktion](http://php.net/manual/de/function.date.php)¹ unterstützt.

Um dies dem Anwender mitzuteilen, ergänzen wir die “Usage Information” entsprechend:

```

1 <?php
2 namespace Helloworld;
3
4 use Zend\ModuleManager\ModuleManager;
5 use Zend\ModuleManager\ModuleEvent;
6 use Zend\Console\Adapter\AdapterInterface as Console;
7 use Zend\ModuleManager\Feature\ConsoleBannerProviderInterface;
8 use Zend\ModuleManager\Feature\ConsoleUsageProviderInterface;
9
10 class Module implements
11     ConsoleBannerProviderInterface, ConsoleUsageProviderInterface
12 {
13     public function getConsoleBanner(Console $console)
14     {
15         return
16             "=====\\n" .
17             "      Helloworld-Modul, Version 1.0      \\n" .
18             "=====\\n";
19     }
20
21     public function getConsoleUsage(Console $console)
22     {
23         return array(
24             'show date [--format=]'
25             => 'Zeigt das aktuelle Datum/Uhrzeit an.',
26
27             array(
```

¹<http://php.net/manual/de/function.date.php>

```

28             '--format=FORMAT',
29             'Es wird die Formatierung ' .
30             'der PHP "date"-Funktion unterstützt.'
31         ),
32     );
33 }
34
35 // [...]
36 }

```

Listing 20.3

Nun erhalten wir die folgende, wirklich hilfreiche Ausgabe:

```

1  $ php index.php
2  > ==-----==
3  >      Helloworld-Modul, Version 1.0
4  > ==-----==
5
6  > index.php show date [--format=]
7  > Zeigt das aktuelle Datum/Uhrzeit an.
8  > --format=FORMAT
9  > Es wird die Formatierung der PHP "date"-Funktion unterstützt.

```

Listing 20.4

20.3 Console-Routen und Request-Verarbeitung

Jetzt geht es an die Implementierung. Zunächst einmal erweitern wird die `module.config.php` des Helloworld-Moduls und eine passende Route:

```

1  <?php
2  // [...]
3  'console' => array(
4      'router' => array(
5          'routes' => array(
6              'date' => array(
7                  'options' => array(
8                      'route' => 'show date',
9                      'defaults' => array(
10                         'controller' => 'Helloworld\Controller\Index',

```

```

11         'action'      => 'date'
12     )
13 )
14 )
15 )
16 )
17 )
18 // [...]

```

Listing 20.5

Dazu ist ein neuer Abschnitt `console` erforderlich, in dem die ‘routes’ definiert werden müssen. Eine “Console-Route” wird also nicht direkt im “Top-Level-Abschnitt” ‘router’ definiert wie gewöhnlich, sondern innerhalb des ‘router’-Abschnitts im “Top-Level-Abschnitt” ‘console’. Ein feiner, aber wichtiger Unterschied. Die eigentliche Definition ist im Grunde selbsterklärend, wenn man sich bereits ein wenig mit dem Mechanismus des Routings im Zend Framework 2 beschäftigt hat. Nichts Neues hier also.

Die eigentliche Action ist auch sehr überschaubar und nicht besonderes:

```

1  <?php
2  public function dateAction()
3  {
4      return date('D M d H:i:s e Y') . PHP_EOL;
5  }

```

Listing 20.6

Der Format-String `D M d H:i:s e Y` sorgt nun dafür, dass die Ausgabe dem Ergebnis des `date`-Kommandos entspricht (zumindest hier exemplarisch auf meiner Shell):

```

1  $ date
2  > Fri Aug 31 15:48:18 UTC 2012
3  $ php index.php show date
4  > Fri Aug 31 15:48:21 UTC 2012

```

Jetzt noch die Sache mit dem “Format-Parameter”. Wenn wir

```

1  $ php index.php show date --format=H:i:s

```

aufrufen, so sehen wir .. das Modul-Banner von oben, aber kein Datum. Das liegt daran, dass unsere zuvor erstellte Route hier nicht “greift”. Wenn wir die Route wie folgt anpassen

```

1  <?php
2  // [...]
3  'console' => array(
4      'router' => array(
5          'routes' => array(
6              'date' => array(
7                  'options' => array(
8                      'route'      => 'show date [--format=]',
9                      'defaults' => array(
10                         'controller' => 'Helloworld\Controller\Index',
11                         'action'      => 'date'
12                     )
13                 )
14             )
15         )
16     ),
17 ),
18 // [...]

```

Listing 20.7

funktioniert der Aufruf:

```

1  $ php index.php show date --format=H:i:s
2  > 18:43:38

```

Grundsätzlich gibt es eine ganze Reihe von Möglichkeiten, wie dynamische Bestandteile eines Konsolenaufrufs übergeben werden können:

- “Value Flag Parameter”: Haben wir bereits oben mit “--format=” eingesetzt. Durch die eckigen Klammern haben wir den Parameter zudem optional gemacht. Das Vorteil des “Value Flag Parameter” ist die Tatsache, dass die Reihenfolge der einzelnen Parameter bei mehreren Parametern egal ist. Das entspricht dem üblichen Verhalten von Shell-Kommandos, bei denen ebenfalls die Reihenfolge fast immer egal ist. Im Controller kann man wie folgt auf den übergebenen Wert zugreifen:

```

1  <?php $this->getRequest()->getParam('format');

```

Listing 20.8

- “Positional Value Parameter”: Wenn wir den Parameter als “integralen Bestandteil” des Kommandos hätten realisieren wollen (anstatt über ein “Flag”), so hatten wir einen “Positional Value Parameter” einsetzen können. Allerdings ist dabei - der Name verrät es schon - die

Position des Parameters im Kommando wichtig. Die Routen-Definition müsste dann wie folgt aussehen: 'show date [<format>]'. Das Zentrale sind hier die spitzen Klammern; über die umgebenen eckigen Klammern wird analog die Optionalität gesteuert. Die Action könnten wir dann wie folgt implementieren:

```

1  <?php
2  public function dateAction()
3  {
4      if($this->getRequest()->getParam('format'))
5          return date($this->getRequest()->getParam('format')) . PHP_EOL;
6      else
7          return date('D M d H:i:s e Y') . PHP_EOL;
8  }
```

Listing 20.9

- “Literal Parameter”: Wir hätten den “Format-String” auch mit Hilfe eines “Literal Parameter” als Bestandteil des eigentlichen Kommandos ausgestalten können. Allerdings nur dann, wenn es eine im Vorfeld definierte Menge potenzieller Formate gegeben hätte. Über einen “Literal Parameter” liesse sich etwa elegant steuern, ob nur das Datum oder zusätzlich auch die Uhrzeit ausgegeben werden soll. Dazu würde man die “Routen-Definition” wie folgt gestalten: 'show date [time]'. Auf diese Weise hätten die folgenden Kommandos die gezeigten Ergebnisse:

```

1  $ php index.php show date time
2  > Sat Sep 01 10:20:10 UTC 2012
3  $ php index.php show date
4  > Sat Sep 01 UTC 2012
```

Die Action dazu könnten wir wie folgt implementieren:

```

1  <?php
2  public function dateAction()
3  {
4      $result = date('D M d');
5
6      if($this->getRequest()->getParam('time'))
7          $result .= date(' H:i:s');
8
9      return $result . date(' e Y') . PHP_EOL;
10 }
```

Listing 20.10

- “Value Flag”: Eine besondere Form ist das “Value Flag”, das dem “Value Flag Parameter” ähnlich ist, es allerdings nicht ermöglicht, zusätzlich noch einen Wert zu übergeben. Ein “Value Flag” beeinflusst die Verarbeitung bereits durch seine bloße Anwesenheit. Es bietet sich z.B. für die Aktivierung eines Debug-Modus via “-debug” an. Lässt man im Gegensatz zum “Value Flag Parameter” in der Routen-Definition das “=” am Ende weg, wird das Flag als “Value Flag” interpretiert und es kann in einer Action wie folgt auf dessen Existenz geprüft werden:

```
1 <?php if($this->getRequest()->getParam('debug'));
```

Listing 20.11

Sicherlich ist das jetzt nicht “wasserdicht” implementiert, aber es tut grundsätzlich mal seinen Dienst. Das soll uns hier für den Moment reichen. Wenn wir sicherstellen wollen, dass die Action über die Konsole aufgerufen wird, können wir dies übrigens durch Prüfung des Request-Typs realisieren:

```
1 <?php
2 if (!$request instanceof ConsoleRequest){
3     throw new RuntimeException(
4         'You can only use this action from a console!'
5     );
6 }
7
8 **Listing 20.12**
```

Üblicherweise wird man bei Kommandos auf der Kommandozeile für eine Korrelation zwischen Controller, Action und dem entsprechenden Kommando sorgen wollen. `show date` wäre in einem echten Szenario vermutlich auf einen `DateController` mit dessen `dateAction()` abgebildet - muss es aber nicht, wie wir gerade gesehen haben, denn grundsätzlich ist man bei der Definition der Kommandos wieder völlig frei.



Interaktive Kommandozeile

Über `Zend\Console` kann die Kommandozeile bei Bedarf sogar auch interaktiv gestaltet werden, etwa um Parameterwerte abzufragen, die nicht bereits im Rahmen des Kommandos übergeben wurden.

21 Internationalisierung

Das World Wide Web hat eine faszinierende Eigenschaft: Jedes Online-Produkt, jedes Web-Startup, ist direkt mit dem Launch ein weltweit verfügbares, globales agierendes Konstrukt. Ob man nun will, oder nicht. Die Frage ist eigentlich nur, wie einfach man es internationalen Nutzern macht, die Anwendung zu verwenden.

Das Zend Framework 2 bringt einen komplett neu geschriebenen Internationalisierung-Layer (i18n) mit, zumindest wird er offiziell so bezeichnet. Er basiert auf der PHP Extension “INTL”, die seit PHP 5.3.0 verfügbar ist. Die “INTL”-Extension selbst ist ein Wrapper für die [ICU¹](#)-Bibliothek, deren Einsatz bereits seit einigen Jahren in der C/C++ und Java-Welt üblich ist und die einen Programmiersprachen übergreifenden Standard für die Internationalisierung von Anwendungen darstellt. Wenn man ehrlich ist, trifft es die Bezeichnung “Layer” nicht richtig, handelt es sich bei Zend\I18n doch vielmehr um eine Reihe von Hilfsfunktionen, die letztlich immer sehr direkt von der “INTL”-Extension gebrauch machen, wie wir auch später noch sehen werden.

Eben weil die i18n-Funktionen des Frameworks auf der “INTL”-Extension basieren, muss diese auf jeden Fall im System verfügbar sein. Andernfalls können die Funktionen nicht eingesetzt werden. Mit Hilfe von `php -i` auf der Kommandozeile (oder wahlweise natürlich auch via `phpinfo()`) kann einfach herausgefunden werden, ob die Extension bereits verfügbar ist oder erst noch installiert werden muss:

```
1 $ php -i | grep intl
2 > /etc/php5/cli/conf.d/intl.ini,
3 > intl
```

Hier sieht alles gut aus. Wenn die Bibliothek nachträglich installiert werden muss, so sollte man unbedingt daran denken, im Anschluss den Webserver neu zu starten. Andernfalls bleibt die Bibliothek samt all seiner Klassen zunächst verborgen und man fragt sich, wieso der eigene Code denn trotzdem noch immer nicht läuft.

Alle im Folgenden beschriebenen Funktionen zur Internationalisierung basieren maßgeblich auf der sog. “Locale”, die Wikipedia wie folgt umschreibt:

“Eine Locale ist ein Einstellungssatz, der die Gebietsschemaparameter für Computerprogramme enthält. Dazu gehören in erster Linie die Sprache der Benutzeroberfläche, das Land [...] Zahlen-, Währungs-, Datums- und Zeitformaten. Ein Einstellungssatz wird üblicherweise mit einem Code, der meist Sprache und Land umfasst, eindeutig identifiziert.” (Wikipedia)

¹<http://www.php.net/manual/de/intro.intl.php>

Dieser Code hat meist die Gestalt von “sprache LAND”, also etwa de_DE für Deutsch/Deutschland oder de_AT für Deutsch/Österreich. Seltener wird statt dem Unterstrich auch der Bindestrich zur Trennung von Sprache und Land verwendet. An diesem Code “hängen” eine ganze Reihe von Einstellungen, wie wir im Folgenden sehen werden.

Übrigens unterscheidet man üblicherweise zwischen “Internationalisierung (i18n)” und “Lokalisierung (L10n)”. Es mag aus akademischer Sicht Unterschiede zwischen den Bezeichnungen geben, in der Praxis werden sie aber meist doch synonym verwendet. Meint man doch fast immer die

“Anpassung von [...] Computerprogrammen (Software) an die in einem bestimmten geographisch oder ethnisch umschriebenen Absatz- oder Nutzungsgebiet (Land, Region oder ethnische Gruppe) vorherrschenden lokalen sprachlichen und kulturellen Gegebenheiten.” (Wikipedia)

Das [W3C²](http://www.w3.org/), als eine der wichtigsten Organisationen des World Wide Web, hat mal gesagt, das “Internationalisierung” bedeutet, “Lokalisierung” auf einfache Art zu ermöglichen. Wenn man also so will, betreibt man etwa damit, dass man “hartverdrahtete” Text-Schnipsel, z.B. in Templates, in dynamisch zu ladende “Sprachdateien” auslagert, “Internationalisierung” im engeren Sinne. Während die eigentliche Übersetzung ebenjener Text-Schnipsel in eine bestimmte Sprache dem Prozess der “Lokalisierung” entspricht. So weit, so akademisch. In der Praxis ist diese Unterscheidung sicherlich weniger wichtig.

21.1 Zahlen-, Währungs- und Datumsformate

Erstellen wir zu Testzwecken eine einfache Action, in der wir die Lokalisierung ausprobieren können

```

1  <?php
2  namespace HelloWorld\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6
7  class IndexController extends AbstractActionController
8  {
9      public function helloAction()
10     {
11         return new ViewModel(
12             array(
13                 "number" => 3324234234.34234243,
14                 "price" => 1039.32,
```

²<http://www.w3.org/>

```

15         "date" => new \DateTime(),
16         "greeting" => "Willkommen"
17     );
18 };
19 }
20
21 // [...]
22 }

```

Listing 21.1

sowie eine passende Route:

```

1  <?php
2  // [...]
3  'hello' => array(
4      'type' => 'Literal',
5      'options' => array(
6          'route' => '/hello',
7          'defaults' => array(
8              'controller' => 'Helloworld\Controller\Index',
9              'action' => 'hello',
10         ),
11     ),
12 ),
13 // [...]

```

Listing 21.2

Im View-File zur Action können wir über den `currencyFormat`-Helper eine sauber formatierte Ausgabe erzeugen:

```

1  <?php
2  echo $this->currencyFormat($this->price, "EUR", "de_DE");

```

Listing 21.3

Das Ergebnis sieht wie folgt aus:

```

1  > 1.039,32 €

```

Lassen wir die Angabe der “Locale” beim Aufruf weg, sehen wir die formatierte Ausgabe auf Basis der “Default Locale”, die, insofern nicht anders definiert, `en_US_POSIX` entspricht (eine spezielle Variante der `en_US`-Locale, die u.a. Stabilität der Definitionen gewährleistet):

```
1 > € 1039.32
```

Statt die “Locale” wie oben bei jedem Aufruf anzugeben, können wir sie vorab auch einmalig setzen:

```
1 <?php
2 class IndexController extends AbstractActionController
3 {
4     private $greetingService;
5
6     public function helloAction()
7     {
8         \Locale::setDefault('de_DE');
9         // [...]
10    }
11    // [...]
12 }
```

Listing 21.4

Die Klasse `Locale` ist übrigens Teil der INTL-Extension und nicht etwa eine Komponente des Frameworks.

Das Datum lässt sich ebenfalls sehr einfach über den entsprechenden View-Helper lokalisiert ausgeben:

```
1 <?php
2 echo $this->dateFormat($this->date);
```

Listing 21.5

Das Ergebnis sieht trotz vorherigem

```
1 <?php
2 \Locale::setDefault('de_DE');
```

Listing 21.6

allerdings doch ein wenig befremdlich aus:

```
1 > 20120902 11:00 vorm.
```

Glücklicherweise bietet der `dateFormat`-Helper, der im Hintergrund auf die INTL-Klasse `IntlDateFormatter` zurückgreift, einige Möglichkeiten zur Anpassung der Formatierung sowohl des Datums (zweiter Parameter), als auch der Uhrzeit (dritter Parameter):

```
1 <?php
2 echo $this->dateFormat(
3     $this->date,
4     \IntlDateFormatter::SHORT,
5     \IntlDateFormatter::SHORT
6 );
```

Listing 21.6

Auch hier ließe sich bei Bedarf zusätzlich als vierten Parameter die betreffende “Locale” übergeben. Über den Aufruf

```
1 <?php
2 echo $this->dateFormat(
3     $this->date,
4     \IntlDateFormatter::SHORT,
5     \IntlDateFormatter::NONE
6 );
```

Listing 21.7

lassen sich auch ganze Bestandteile ausblenden; in diesem Fall der “Uhrzeit”-Anteil. Weitere Formartierungsoptionen finden sich in der [Dokumentation von IntlDateFormatter](#)³.

Auch “normale” Zahlen lassen sich in ihrer Darstellung beeinflussen. So führt

```
1 <?php
2 echo $this->numberFormat($this->number);
```

Listing 21.8

zur Ausgabe von

```
1 > 3.324.234.234,342
```

wenn die Locale de_DE zuvor global gesetzt wurde. Für Zahlen lassen sich der “Style” und der “Format-Typ” definieren, wobei “Format-Typ” etwa maßgeblich beim `currencyFormat`-Helper zum Einsatz kommt, weil dort mit `NumberFormatter::CURRENCY_SYMBOL` die entsprechende Einstellung zur Ausgabe des Währungssymbols vorgenommen wird. Weitere Formartierungsoptionen finden sich in der [Dokumentation von NumberFormatter](#)⁴.

³<http://de.php.net/manual/de/class.intldateformatter.php>

⁴<http://de.php.net/manual/de/class.numberformatter.php>

21.2 Textübersetzungen

Konfiguration des Translator-Service

Neben der lokalisierten Darstellung von Währungen-, Datums- und Zahlenformaten spielen vor allem die Übersetzungen von Textfragmenten in den Masken und Ansichten der Anwendung eine maßgebliche Rolle. Baut man seine Anwendung auf Basis der `ZendSkeletonApplication` auf, findet man in der `module.config.php` des Application-Moduls einen “Top-Level-Abschnitt” `translator` vor:

```
1  <?php
2  // [...]
3  'translator' => array(
4      'locale' => 'de_DE',
5      'translation_file_patterns' => array(
6          array(
7              'type'      => 'gettext',
8              'base_dir' => __DIR__ . '/../language',
9              'pattern'   => '%s.mo',
10         ),
11     ),
12 ),
13 // [...]
```

Listing 21.9

Dies ist die Konfiguration des `Translator(-Service)`, den das Framework mitbringt. Haben wir oben die Konfiguration der “Default Locale” noch direkt in einer Action vorgenommen, ist der in der `ZendSkeletonApplication` beschrittene Weg sicherlich besser, die “Locale” im Rahmen der Modul-Konfiguration festzulegen. Allerdings wird hier lediglich die Locale für den `Translator` festgelegt, über den Textfragmente übersetzt werden können. Die Einstellung hier hat keine Auswirkung etwa auf den `currencyFormat`-Helper. Aber: Wenn man das “Default Locale” einmalig setzt, etwa im Rahmen des Bootstrapping eines Moduls

```

1  <?php
2  class Module
3  {
4      public function onBootstrap($e)
5      {
6          \Locale::setDefault('de_DE');
7          // [...]
8      }
9  }

```

Listing 21.10

richtet sich auch der Translator automatisch danach. Man kann dann also auf die explizite Angabe der “Locale” verzichten:

```

1  <?php
2  // [...]
3  'translator' => array(
4      'translation_file_patterns' => array(
5          array(
6              'type'      => 'gettext',
7              'base_dir' => __DIR__ . '/../language',
8              'pattern'  => '%s.mo',
9          ),
10     ),
11 ),
12 // [...]

```

Listing 21.11

Es empfiehlt sich, die Locale einmalig auf diesem Wege zu setzen, wenn man verschiedene Komponenten von Zend\I18n einsetzen und für einen Request in einer “Locale” bleiben will - was ja meist der Fall ist.

Häufig stellt sich beim Framework die Frage, ob es sich bei einem Key in der Konfiguration um einen “magischen Wert” handelt, der automatisch an irgendeiner Stelle des Bootstrappings vom Framework ausgelesen und verarbeitet wird, oder eben nicht. So ist dies ja etwa beim Schlüssel `router` der Fall oder beim `view_manager`. Beim Translator ist es wie folgt: Zwar wird von der `TranslatorServiceFactory`, die das Framework bereitstellt und die also in diesem Fall nicht selbst geschrieben werden muss, der Abschnitt `translator` in der (vereinten) Konfiguration gesucht und automatisch ausgelesen, allerdings wird die Factory selbst nicht standardmäßig vom Framework berücksichtigt. Darum muss man sich demnach noch selbst kümmern bzw. überlässt der `ZendSkeletonApplication` diese Aufgabe:

```
1  <?php
2  // [...]
3  'service_manager' => array(
4      'factories' => array(
5          'translator' => 'Zend\I18n\Translator\TranslatorServiceFactory',
6      ),
7  ),
8  // [...]
```

Listing 21.12

Wenn nun beim `ServiceManager` der `translator`-Service angefragt wird, sorgt dieser dafür, dass die, in diesem Fall vom Framework mitgelieferte `TranslatorServiceFactory`, die oben gezeigte Konfiguration ausliest und den `Translator` damit ausstattet. Die Konfiguration bezieht sich maßgeblich darauf, wie und wo die Übersetzungen abgelegt sind. Grundsätzlich werden die folgenden Arten und Weisen unterstützt, um Textfragmente zu speichern:

- PHP arrays
- gettext
- Tmx
- XLIFF

Die `ZendSkeletonApplication` macht regen Gebrauch von `gettext`⁵, bei dem sog. “po-Textdateien” vorliegen, die mit einem üblichen Editor oder auch spezialisierten Programmen, wie dem Klassiker [Poedit](<http://www.poedit.net/>), geöffnet und bearbeitet werden können und die jeweils sowohl die Keys als auch deren Übersetzung in eine bestimmte Sprache enthalten, sowie die “mo-Binärdateien”, die aus den “po-Textdateien” erzeugt und später von der Anwendung verarbeitet werden. Es gibt bei `gettext` also sozusagen einmal die optimale Datei für den Editor und einmal die optimale, binäre Datei für die Anwendung. Sowohl “Tmx” (Translation Memory eXchange) als auch “XLIFF” (XML Localisation Interchange File Format) sind XML-Anwendungen, die ebenfalls auf die Lokalisierung von Anwendungen hin ausgerichtet sind. Die letztgenannten sind durch das zusätzlich erforderliche XML-Parsing weniger performant. Mindestens bei diesen Lösungen bedarf es eines sinnvollen Cachings.

Über ‘`base_dir`’ wird das Verzeichnis angegeben, in dem die Übersetzungsdateien abgelegt sind. Über ‘`pattern`’ ist definiert, wie das Framework von einer “Locale”, etwa “`de_DE`”, auf das passende File mit den Übersetzungen schließen kann. Ein Mapping also. Zwar gibt es auch andere Varianten, die unterschiedlichen Übersetzungen abzulegen, etwa in einem einzigen File, dennoch empfiehlt sich die Aufteilung in jeweils ein File pro Sprache, so, wie es auch die `ZendSkeletonApplication` macht.

⁵http://de.wikipedia.org/wiki/GNU_gettext

Explizite Übersetzungen

In einer View kann dann je nach zuvor konfigurierter “Locale” über den folgenden Aufruf der passende Inhalt ausgegeben werden:

```
1 <?php
2 $translator->translate($message);
```

Listing 21.13

Auch lassen sich direkt über den translate-Service an beliebiger Stelle lokalisierte Inhalte ausgeben, z.B. führt der Aufruf

```
1 <?php
2 var_dump(
3     $this->getServiceLocator()
4         ->get('translator')
5         ->translate('Home')
6 );
```

Listing 21.14

in einer Action zur Ausgabe von “Startseite”, weil die “ZendSkeletonApplication” im Rahmen ihrer gettext-Dateien die entsprechende Übersetzung für die “Locale” de_DE und den Key Home mitbringt.

Will man einen Text-Key anzeigen, der Platzhalter enthält, die es zu füllen gilt, muss man sich zur Anzeige der sprintf-Funktion von PHP bedienen:

```
1 <?php
2 echo sprintf(
3     $this->translate("We found %d products for you."),
4     $productCount
5 );
```

Listing 21.15

Führt zu einer Ausgabe von

```
1 > Wir haben 12 Produkte für Sie gefunden.
```

insofern die entsprechende Übersetzung hinterlegt und die Variable \$productCount auf den Wert 12 gesetzt ist. Der Translator kümmert sich also nicht selbst um das Füllen von Platzhaltern.

Formulare, Validatoren & Co.

Damit auch Texte, die nicht explizit via `translate()` ausgegeben und übersetzt werden, passend lokalisiert dargestellt werden, sollten noch einige weitere Vorbereitungen getroffen werden.

Da wären etwa die “Validatoren”, die etwa im Rahmen von Formularen Fehlermeldungen erzeugen können. Damit diese Nachrichten lokalisiert werden können, empfiehlt es sich, frühzeitig den Translator auch für die Validatoren zu registrieren. Dies kann man z.B. in der `Module`-Klasse an der Stelle machen, an der man auch die (zuvor ermittelte) `Locale` setzt:

```
1  <?php
2  class Module
3  {
4      public function onBootstrap($e)
5      {
6          \Locale::setDefault('de_DE');
7          \Zend\Validator\AbstractValidator::setDefaultTranslator(
8              $e->getApplication()
9                  ->getServiceManager()
10                     ->get('translator')
11          );
12
13      // [...]
14  }
```

Listing 21.16

Auch über “View Helper” werden häufig indirekt Ausgaben erzeugt, etwa bei Verwendung der “Form View Helper”:

```
1  <?php
2  echo $this->formLabel($this->form->get('username'));
```

Listing 21.17

Um dafür zu sorgen, dass automatisch lokalisierte Inhalte ausgegeben werden, muss man hier statt einer expliziten Angabe, wie noch zuvor beim `Validator`, eine Konvention einhalten, nämlich die, dass ein fertig konfigurierter `Translator` unter dem Key `translator` im `ServiceManager` zu finden ist:

```

1  <?php
2  // [...]
3  'service_manager' => array(
4      'factories' => array(
5          'translator' => 'Zend\I18n\Translator\TranslatorServiceFactory',
6      ),
7  ),
8  // [...]

```

Listing 21.18

Diese Tatsache macht sich dann, wenn ein “View Helper” angefordert wird, der `Zend\View\HelperPluginManager` zu nutze, der nach dem `translator` im `ServiceManager` Ausschau hält und - insofern vorhanden - automatisch in den jeweiligen “View Helper” injiziert, der ihn dann verwendet:

```

1  <?php
2  public function injectTranslator($helper)
3  {
4      if ($helper instanceof TranslatorAwareInterface) {
5          $locator = $this->getServiceLocator();
6
7          if ($locator && $locator->has('translator')) {
8              $helper->setTranslator($locator->get('translator'));
9          }
10     }
11 }

```

Listing 21.19

Anwender des Zend Framework 1 wird das bekannt vorkommen, war es seinerzeit doch sehr ähnlich geregelt, nur kam da noch die `Zend_Registry` zum Einsatz und der Key musste auf `Zend_Translate` lauten:

```

1  <?php
2  Zend_Registry::set('Zend_Translate', $translator);

```

Listing 21.20



Text-Domains zur Verwaltung von Sprachkeys

Mit Hilfe von Sprachkey-Kategorien, sog. “Text-Domains”, lassen sich bei Bedarf Sprachkeys innerhalb von Übersetzungsdateien der besseren Übersicht halber gruppieren.



Bereits übersetzte Fehlermeldungen

Das Framework bringt dankenswerter Weise bereits Übersetzungen für die Fehlermeldungen der Standard-Validatoren in mehr als 40 Sprachen mit. Sie finden sich im Verzeichnis `resources/languages` und können über die Methode `addTranslationFile()` des `Zend\I18n\Translator\Translator` verfügbar gemacht werden.

21.3 Ermitteln und Konfigurieren der Locale

Stellt sich eigentlich nur noch die Frage, wie man nun an die Locale kommt, die man gerade für einen Request einstellen muss? Wenn man sie “hartverdrahtet”, so wie in der `ZendSkeletonApplication`, so ist der Mehrwert einer internationalisierten Anwendung ja nahezu Null. Welche Mittel gibt es also, die Locale zu bestimmen? Die folgenden Ansätze bieten sich u.a. etwa dafür an:

- Auswerten der Top-Level-Domain: So etwa macht es eBay oder Amazon. Wenn man `eBay.de` öffnet, so sieht man klar, dass das Locale `de_DE` aktiviert ist. Wechselt man zu `eBay.com`, so ist das Locale `en_US` aktiv. Auch, wenn sehr wohl erkannt wird, dass ich aus Deutschland die Seite besuche. Ich sehe das etwa daran, dass mir deutschsprachige Werbung angezeigt wird.
- Auswerten der Subdomain: So macht es `Shopping.com` durch Verwendung von `de.shopping.com` und `uk.shopping.com`.
- Auswerten des Pfades: Für den Fall, dass die Locale über den Pfad einer URL transportiert wird.
- Browsereinstellungen verwenden: Üblicherweise sendet ein Client bei einem Request den `HTTP_ACCEPT_LANGUAGE`-Header auf Basis der Browsereinstellungen mit. Auch dieser Wert (bzw. die Werte) kann (bzw. können) herangezogen werden.
- Geo-Targeting: Über die IP-Adresse des Nutzers oder auch über die HTML5 Geolocation API lässt sich die gewünschte Locale auf Basis des Aufenthaltsorts des Nutzers bestimmen.
- Benutzerprofil, aktive Auswahl, Cookies: Zusätzlich zu den eher grundsätzlichen Ansätzen kann natürlich auch die Anwendung dafür sorgen, dass ein Nutzer aktiv die “Locale” bestimmt und diese kann dann im Rahmen einer Session oder eines Cookies für die Anwendung verfügbar ist.

Welche dieser Optionen man wählt, hängt vom konkreten Anwendungsfall ab. Die jeweilige Implementierung bietet sich für die `onBootstrap()`-Methode in der `Module`-Klasse eines der Anwendungsmodule an. Weil dort das `MvcEvent`-Objekt bereitgestellt wird, lässt sich leicht auf den Request zugreifen und so die notwendigen Informationen auslesen:

```
1  <?php
2  class Module
3  {
4      public function onBootstrap($e)
5      {
6          var_dump($e->getRequest());
7      }
8
9      // [...]
10 }
```

Listing 21.21

Wer sich die Mühe sparen will, selbst die passende Lösung zu implementieren, der sollte einen Blick auf das Modul [SlmLocale](https://github.com/juriansluijman/SlmLocale)⁶ werfen, dass bereits viele der oben beschriebenen Strategien unterstützt. Und das bei Bedarf auch in Kombination.

⁶<https://github.com/juriansluijman/SlmLocale>

22 AJAX & Webservices

22.1 AJAX mit JSON

Auch wenn sich die Überschrift dieses Abschnitt in gewisser Weise selbst widerspricht - das “X” in AJAX steht schließlich für XML und nicht für JSON - so hat sich das Datenformat JSON in den letzten Jahren doch gegen seinen schwergewichtigen Konkurrenten XML für AJAX-Anwendungen durchgesetzt. Die “JavaScript Object Notation”, kurz [JSON¹](http://www.json.org/), ist demnach ein eher leichtgewichtiges Datenformat für den Austausch von Informationen zwischen Maschine und Maschine, wobei die Inhalte aufgrund der Textorientierung auch für den Menschen lesbar sind.

Um beim Aufruf einer URL eine JSON-Datenstruktur zurückzugeben, die dann etwa im Rahmen eines AJAX-Calls im Client verarbeitet werden kann, muss man in der jeweiligen Action lediglich dafür sorgen, dass das von der Action erzeugte “View Model” von Typ `JsonModel` ist

```
1  <?php
2  namespace HelloWorld\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\JsonModel;
6
7  class AuthController extends AbstractActionController
8  {
9      public function helloAction()
10     {
11         $result = new JsonModel(
12             array(
13                 'greeting' => 'hello world',
14             )
15         );
16
17         // [...]
18     }
```

Listing 22.1

und in einem der `module.config.php`-Dateien der `ViewManager` dahingehend sensibilisiert wird, dass er ein “View Model” vom Typ `JsonModel` auch korrekt verarbeitet:

¹<http://www.json.org/>

```
1  <?php
2  // [...]
3  'view_manager' => array(
4      // [...]
5      'strategies' => array(
6          'ViewJsonStrategy',
7      ),
8  ),
9  // [...]
```

Listing 22.2

So wird dann der korrekte Content-Typ gesetzt und auch das Rendering eines etwaig konfigurierten Layouts unterbunden.

Arbeitet man viel mit JSON-Datenstrukturen, die auf dem Server generiert und im Client verarbeitet werden, hat man es meistens auch mit viel Client-Code zu tun. Müssen die Datenstrukturen doch im Client verarbeitet und irgendwann letztlich auch in (HTML-)Markup überführt werden. Häufig verwendet man zusätzlich zum Zend Framework 2 mit dessen MVC-Komponente auch im Client MVC-basierte Bibliotheken, die das Leben weiter erleichtern. Während bei klassischen Websites dann JS-Frameworks wie [Backbone.js](http://backbonejs.org/)² zum Einsatz kommen, sind es z.B. bei mobilen Clients die Bibliotheken der jeweiligen Plattformen. So oder so ist der Entwicklungsaufwand im Client bei diesem Ansatz - insofern man ihn in einem größeren Stile betreibt - verhältnismäßig hoch. Dafür bietet er die größtmögliche Flexibilität und insbesondere auch gute Unterstützung für Endgeräte, deren UI-Darstellung nicht über Webtechnologien realisiert wird.

22.2 AJAX mit HTML

Alternativ dazu bietet es sich an, nicht nur “nackte” Datenstrukturen zu übertragen, sondern bereits fertig erzeugte “HTML-Fragmente”, sog. “Partials”, die im Client dann nur noch 1-zu-1 an die richtige Stelle im DOM “eingehängt” werden müssen. Auf diese Art und Weise erspart man sich die Arbeit, die ansonsten noch erforderlich ist, um die Daten im Client in etwas “ansehnliches” zu überführen. Auch die Arbeit mit “Partials” ist zugegebenermaßen wieder nicht AJAX im engeren Sinne - bei dem ja traditionell XML zu Einsatz kommt - aber doch sehr praktisch, minimiert sie doch die Entwicklungsaufwände, die man zusätzlich sonst noch im Client hätte.

Um “AJAX mit HTML” zu praktizieren, verhindert man bei den entsprechenden Actions, dass das Layout der Anwendung berücksichtigt und lediglich das Ergebnis der Action zurückgeliefert wird:

²<http://backbonejs.org/>

```

1  <?php
2  $result = new ViewModel(
3      array(
4          'greeting' => 'hello world!',
5      )
6  );
7
8  $result->setTerminal(true);
9  return $result;

```

Listing 22.3

Gibt man in dem zur Action passenden Template den Wert aus

```

1  <h2><?php echo $this->greeting; ?></h2>

```

Listing 22.4

so sieht man wie erwartet die Überschrift

```

1  hello world

```

auf dem Bildschirm. Wenn man dieses HTML-Fragment nun über einen AJAX-Call lädt, kann man es 1-zu-1 in das DOM einhängen und somit den Inhalt der Seite dynamisch verändern, ohne sie vollständig neu laden zu müssen.

Ruft man den Host ohne weitere Pfad-Information auf, so zeigt sich üblicherweise die Startseite der ZendSkeletonApplication. Wenn wir deren Navigation einmal dahingehend verändern, dass wir einen weiteren Menüpunkt “Greeting” hinzufügen und so verlinken, dass bei Klick via AJAX die Ausgabe von oben anstelle der sog. “Hero Unit” - der großen Box ganz oben auf der Seite - angezeigt wird

```

1  <ul class="nav">
2      <li class="active">
3          <a href="<?php echo $this->url('home') ?>">
4              <?php echo $this->translate('Home') ?>
5          </a>
6      </li>
7      <li>
8          <a href="javascript:$('.hero-unit').load(
9              'http://localhost:8080/sayhello');">
10             Greeting
11          </a>
12      </li>
13 </ul>

```


Listing 22.5

funktioniert das schon ganz gut. Bei Klick wird die “Hero Unit” durch unsere Begrüßung ersetzt. Vorausgesetzt natürlich, die passende Action ist zuvor unter dem Pfad `/sayhello` verfügbar gemacht worden. Ich verwende in diesem Beispiel übrigens [jQuery](http://jquery.com/)³ für den AJAX-Call und die DOM-Manipulation. Weil die `ZendSkeletonApplication` auf [Bootstrap](http://twitter.github.com/bootstrap/)⁴ basiert, bringt sie auch jQuery gleich mit.

Selbstverständlich ist die Implementierung des jQuery-orientierten JavaScripts hier nicht sonderlich gut gelöst. Es ging mir aber auch nur um das Beispiel, nicht um atemberaubende Frontend-Entwicklung. Daran sieht man aber sehr schön, das sich der Ansatz “AJAX mit HTML” gerade dann auch sehr gut eignet, wenn man selbst, bzw. ein in die Entwicklung involviertes Team, die Kompetenzen “auf dem Server” und beim Umgang mit PHP sieht. Hält man sich strikt an dieses Vorgehen, so schreibt man viel mehr PHP- und weniger JavaScript-Code und schützt sich damit auch ein wenig selbst davor, auf dem Client umfangreichen “Spaghetti-Code” zu produzieren, weil man dort vielleicht weniger Kompetenz oder Erfahrung besitzt.

Will man diesen Pfad weiter verfolgen, bietet sich unbedingt ein Blick auf [pjax](http://pjax.heroku.com/)⁵ an, dass auch als [jQuery-Plugin bei GitHub](https://github.com/defunkt/jquery-pjax)⁶ verfügbar ist. Das gute bei pjax ist die Tatsache, dass es neben AJAX auch “HTML5 pushState” realisiert und zusätzlich zum Laden der Nutzdaten auch noch die Browser-History entsprechend modifiziert. So lassen sich einfach AJAX-Anwendungen entwickeln, die zu den üblichen “Browser-Controls” wie “Zurück” oder “Bookmark” kompatibel sind und sich wegen den geringeren Datenmengen und vermiedenen “Page-Reloads” deutlich flinker in der Interaktion anfühlen.

22.3 RESTful-Webservices

Wenn man in einem Satz sagen soll, was genau ein “RESTful Webservice” eigentlich ist, kommt man schnell ins schwimmen. Ich sage dann immer: Ein “RESTful Webservice” ist eine Website, die für die Nutzung durch Maschinen gemacht ist. Und tatsächlich stützt sich [REST](http://de.wikipedia.org/wiki/Representational_State_Transfer)⁷ ja auf viele Kerngedanken einer “normalen Website”:

- Verwendung von Hypermedia: Die erzeugten Datenstrukturen (z.B. HTML, XML oder JSON) können Links zu anderen Ressourcen enthalten.
- (Ausschließliche) Nutzung von HTTP
- Nutzung “gewöhnlicher” URLs
- Caching von Inhalten mit “Bordmitteln” möglich, etwa über Proxy-Systeme in der Internet-Infrastruktur

³<http://jquery.com/>

⁴<http://twitter.github.com/bootstrap/>

⁵<http://pjax.heroku.com/>

⁶<https://github.com/defunkt/jquery-pjax>

⁷http://de.wikipedia.org/wiki/Representational_State_Transfer

Bei REST dreht sich zudem alles um die Ressourcen, bzw. die Entities einer Anwendung, die über die standardmäßigen Aktionen, die von HTTP definiert werden - also etwa GET, POST, PUT und DELETE - manipuliert werden. Durch diese vereinheitlichte Schnittstelle (bezogen auf die angebotenen Funktionen) wird die Anbindung so eines Webservices enorm vereinfacht. SOAP-Webservices gehen da einen vollkommen anderen Weg, wie wir später noch sehen werden.

Aufgrund der Natur von REST-Webservices bietet sich die direkte Integration in die MVC-Implementierung des Zend Framework 2 an. Mit dem `AbstractRestController` bringt das Framework auch gleich hilfreiche Unterstützung mit.

Um einen “RESTful Webservice” zu erzeugen, legen wir zunächst einen Controller vom Typ `AbstractRestController` an, den wir `ProductController` nennen:

```
1  <?php
2  namespace Helloworld\Controller;
3
4  use Zend\Mvc\Controller\AbstractRestController;
5  use Zend\View\Model\JsonModel;
6
7  class ProductController extends AbstractRestController
8  {
9      public function getList()
10     {}
11
12     public function get($id)
13     {}
14
15     public function create($data)
16     {}
17
18     public function update($id, $data)
19     {}
20
21     public function delete($id)
22     {}
23 }
```

Listing 22.6

Der `AbstractRestController` macht es erforderlich, dass wir die oben gezeigten Methoden implementieren, um die “Basisfunktionalität” eines “RESTful Webservice” abzubilden. Im Gegensatz zum `AbstractActionController`, der im Rahmen der “normalen” Requestverarbeitung zum Einsatz kommt, hält der `AbstractRestController` für das “Dispatching” nicht Ausschau nach dem Parameter `action`, der ansonsten im Rahmen des Routings ermittelt wird (obwohl er auch das

tun würde, insofern erforderlich), sondern er wirft einen Blick auf die HTTP-Methode, die für den Request verwendet wurde, also GET, POST oder ähnliches. Je nach dem, welchen Wert er erhält, ruft er dann die passende Methode in dem Controller auf. Eine Besonderheit gibt es bei GET: Wenn der Parameter `id` ermittelt werden kann, entweder als Teil des Query-Strings oder gesetzt im Rahmen des Routings, so wird die Methode `get()` aufgerufen, ansonsten `getList()`.

Wenn wir nun eine passende Route anlegen (man beachte den fehlenden “action”-Key), z.B.

```

1  <?php
2  // [...]
3  'restful-products' => array(
4      'type'      => 'Literal',
5      'options'   => array(
6          'route'   => '/rest/product',
7          'defaults' => array(
8              'controller' => 'Helloworld\Controller\Product'
9          ),
10     ),
11 ),
12 // [...]
```

Listing 22.7

so können wir die `getList()`-Methode schon einmal wie folgt implementieren:

```

1  <?php
2  namespace Helloworld\Controller;
3
4  use Zend\Mvc\Controller\AbstractRestfulController;
5  use Zend\View\Model\JsonModel;
6
7  class ProductController extends AbstractRestfulController
8  {
9      private $products = array(
10         array(
11             'name' => 'Boxfresh SPARKO 4 Sneaker',
12             'price' => 84.95
13         ),
14         array(
15             'name' => 'adidas Originals Samba',
16             'price' => 64.95
17         )
18     );
```

```

19
20     public function getList()
21     {
22         return new JsonModel(
23             $this->products
24         );
25     }
26
27     // [...]
28 }

```

Listing 22.8

Wobei wir hier zu Demonstrationszwecken auf Produktdaten zugreifen, die wir in Form eines PHP-Arrays vorhalten - normalerweise würde man hier vermutlich auf Datenbanken zugreifen und mit Entities hantieren - und mit einem JSON-Datenformat als Ergebnis arbeiten.

Wenn wir nun die URL `/rest/product` aufrufen, so sehen wir:

```

1  [
2      {
3          "name": "Boxfresh SPARKO 4 Sneaker",
4          "price": 84.95
5      },
6      {
7          "name": "adidas Originals Samba",
8          "price": 64.95
9      }
10 ]

```

Wenn wir die URL mit `id` aufrufen, also etwa in der Form `/rest/product?id=1`, so wird automatisch nicht mehr die `getList()`-Methode, sondern die `get()`-Methode aufgerufen, die wir hier etwas naiv wie folgt implementieren könnten:

```

1  <?php
2  public function get($id)
3  {
4      return new JsonModel(
5          $this->products[$id-1]
6      );
7  }

```

Listing 22.9

Die Ausgabe sieht dann so aus:

```
1 {"name":"Boxfresh SPARKO 4 Sneaker","price":84.95}
```

Auch die Erzeugung eines Produkts können wir leicht implementieren. Die größte Herausforderung ist dabei noch die Tatsache, dass wir dazu einen POST-Request absenden müssen, was mit den Bordmitteln eines Browsers gar nicht so einfach ist. Für Anwender des Google Chrome Browser empfiehlt sich daher die Extension [Postman](#)⁸, die genau für diesen Zweck erstellt wurde und sich hervorragend dazu eignet, “RESTful Webservices” anzusprechen.

Ich erzeuge also mit Postman einen POST-Request an `/rest/product`, bei dem ich die Parameter “name” und “price” jeweils mit den Werten “Gola QUOTA” und “82.98” übertrage. Um das Ergebnis direkt zu sehen, gebe ich als Ergebnis des Aufrufs die gesamte, um das entsprechende Produkt erweiterte Produkt-Liste zurück:

```
1 <?php
2 public function create($data)
3 {
4     $this->products[] = $data;
5
6     return new JsonModel(
7         $this->products
8     );
9 }
```

Listing 22.10

Der `AbstractRestController` stellt der `create`-Methode die POST-Daten dankenswerterweise direkt als Methodenparameter zur Verfügung.



SOAP, XML-RPC & Co.

Neben REST bringt Zend Framework 2 auch Unterstützung etwa für SOAP und XML-RPC, jeweils sogar für die Server- und Clientseite, mit.

⁸<http://goo.gl/VgmkD>

23 E-Mails versenden

Mit `Zend\Mail` bringt das Framework Unterstützung für den Versand von E-Mails mit. Zwar hat PHP mit der `mail()`-Funktion auch selbst bereits alles notwendige “an Bord”, um E-Mails versenden zu können, allerdings wird die Erstellung und der Versand von E-Mails mit `Zend\Mail` objektorientiert abgebildet und dadurch noch einmal ein gutes Stück vereinfacht, insbesondere bei “gehobenen Ansprüchen”, wie etwa Multipart-E-Mails, wie wir im Folgenden noch sehen werden.

23.1 Eine E-Mail versenden

Eine E-Mail zu versenden ist einfach. Es wird lediglich ein `Message`-Objekt und ein “Transport” benötigt:

```
1 <?php
2 $mail = new \Zend\Mail\Message();
3 $mail->setBody('Hi, Michael!');
4 $mail->setFrom('sender@example.com', 'Sender name');
5 $mail->addTo('zfbuch@michael-romer.de', 'Michael Romer');
6 $mail->setSubject('Mail subject');
7 $transport = new \Zend\Mail\Transport\Sendmail();
8 $transport->send($mail);
```

Listing 23.1

Der “Transport” bestimmt, wie die Nachricht physikalisch Übertragen wird. Für die folgenden Varianten bringt das Framework gleich die entsprechende Implementierung mit. Weitere können bei Bedarf auf Basis des `TransportInterface` vom Anwendungsentwickler programmiert werden.

- **Sendmail:** Der Versand erfolgt schlussendlich über die PHP-Funktion `mail()`, welche wiederum ggf. eine lokale Sendmail-Installation (oder etwas dazu Kompatibles) verwendet.
- **Smtp:** Wie der Name bereits verrät, wird bei diesem “Transport” ein sog. “Message Transfer Agent (MTA)” bemüht, zu dem eine Socket-Verbindung aufgebaut wird. Es werden dazu bei Bedarf verschiedene Arten der Authentifikation unterstützt.
- **File:** Die Nachricht wird nicht tatsächlich versendet, sondern in Form einer Datei in einem definierten Verzeichnis abgelegt. Dieser “Transport” ist etwa in Entwicklungs- und Testsystemen sehr hilfreich.

23.2 Eine HTML-E-Mail versenden

Mit nur geringem Mehraufwand lassen sich auch HTML-E-Mails, bzw. sog. “Multipart-E-Mails” versenden, die sowohl von HTML-fähigen E-Mail-Clients, als auch von rein textbasierten Tools korrekt verarbeitet werden können:

```
1  <?php
2  $textPart = new \Zend\Mime\Part('Hi, Michael!');
3  $textPart->type = "text/plain";
4
5  $htmlPart = new \Zend\Mime\Part("<h1>Hi, Michael!</h1>");
6  $htmlPart->type = "text/html";
7
8  $body = new \Zend\Mime\Message();
9  $body->setParts(array($textPart, $htmlPart));
10
11 $mail = new \Zend\Mail\Message();
12 $mail->setBody($body);
13 $mail->setFrom('sender@example.com', 'Sender name');
14 $mail->addTo('zf2buch@michael-romer.de', 'Michael Romer');
15 $mail->setSubject('Mail subject');
16
17 $mail->getHeaders()->get('content-type')
18     ->setType('multipart/alternative');
19
20 $transport = new \Zend\Mail\Transport\Sendmail();
21 $transport->send($mail);
```

Listing 23.2

Zunächst werden hier die beiden Teile der Nachricht (daher auch die Bezeichnung “Multipart-E-Mail”) erstellt: Einmal der Text-Teil, einmal der HTML-Teil. Beide zusammen bilden den “Body” der E-Mail. Nicht vergessen, über den Header content-type dem Client bei der Interpretation der Daten zu helfen.

23.3 Templates für HTML-E-Mails einsetzen

In aller Regel kommt in einer HTML-E-Mail etwas komplexeres Markup zum Einsatz, als in dem Beispiel oben. Um dann weiter die Übersicht zu behalten und die Trennung der unterschiedlichen Belange des E-Mail-Versands sicherzustellen, empfiehlt sich der Einsatz von HTML-Templates. Glücklicherweise lässt sich dazu recht einfach ebenfalls Zend\View einsetzen, das ja auch bereits

im Rahmen der “MVC-basierten-Verarbeitung” von Requests zum Einsatz kommt. Zunächst wird ein Template benötigt:

```
1 <h1>Hi, <?php echo $this->name; ?>!</h1>
```

Listing 23.3

Die eigentliche E-Mail-Erzeugung und der Versand gestaltet sich dann wie folgt (exemplarisch aus einem auf dem `AbstractActionController` basierenden Controller heraus, dem via `$this->getServiceLocator()` der `ServiceManager` zur Verfügung steht):

```
1 <?php
2 // [...]
3 $view = $this->getServiceLocator()->get('ViewManager')->getView();
4
5 $viewModel = new \Zend\View\Model\ViewModel(
6     array('name' => 'Michael')
7 );
8
9 $viewModel->setOption('has_parent', true);
10 $viewModel->setTemplate('helloworld/emails/say-hi-to-user');
11
12 $textPart = new \Zend\Mime\Part('Hi, Michael!');
13 $textPart->type = "text/plain";
14
15 $htmlPart = new \Zend\Mime\Part($view->render($viewModel));
16 $htmlPart->type = "text/html";
17
18 $body = new \Zend\Mime\Message();
19 $body->setParts(array($textPart, $htmlPart));
20
21 $mail = new \Zend\Mail\Message();
22 $mail->setBody($body);
23 $mail->setFrom('sender@example.com', 'Sender name');
24 $mail->addTo('zf2buch@michael-romer.de', 'Michael Romer');
25 $mail->setSubject('Mail subject');
26
27 $mail->getHeaders()->get('content-type')
28     ->setType('multipart/alternative');
29
30 $transport = new \Zend\Mail\Transport\Sendmail();
31 $transport->send($mail);
```


Listing 23.4

Schauen wir der Reihe nach, was hier genau passiert: Zunächst wird über den `ServiceManager` (hier durch sein Interface `ServiceLocator` repräsentiert) der `ViewManager` zur Verfügung gestellt, der uns wiederum die `View` bereitstellt, die auch im Rahmen der Request-Verarbeitung zum Einsatz kommt (und damit ja standardmäßig auf den `PhpRenderer` zurückgreift, der das oben gezeigte Template verarbeiten kann). Dann wird ein `ViewModel` erzeugt, das die dynamischen Werte hält, die wir jeweils mit dem E-Mail-Template verquicken wollen.

Jetzt kommt ein Kniff: Über die Option `has_parent` sorgen wir dafür, dass später des Ergebnis des Renderings als Rückgabewert der Methode `render()` zur Verfügung gestellt wird, damit wir es als HTML-Teil der E-Mail verwenden können. Der Hintergrund dieses Verhaltens ist wie folgt und hat mit dem, was wir erreichen wollen, eigentlich gar nichts zu tun: Üblicherweise wird das Ergebnis des View-Renderings direkt in das Response-Objekt injiziert, das später der Aufrufer im Rahmen der MVC-Anfrageverarbeitung zurückgegeben bekommt. Dies ist allerdings nicht der Fall, wenn das jeweilige `ViewModel`, das für das Rendering herangezogen wird, nicht das “Top-Level”-`ViewModel` ist, etwa dann, wenn mit einem Layout gearbeitet wird. Dann nämlich bekommt das jeweilige `ViewModel` automatisch die Option `has_parent = true` durch das Framework und das Ergebnis des Renderings wird nicht 1-zu-1 Teil der Response, sondern in “Rohform” zur weiteren Verarbeitung, etwa durch das Framework selbst, bereitgestellt. Hier machen wir uns dieses Verhalten zu Nutze, denn wir benötigen ja ebenjene “Rohform” des Ergebnis des Renderings für unsere E-Mail.

Danach erstellen wir die beiden Teile der Multipart-E-Mail, wobei der HTML-Teil inhaltlich dem Ergebnis des Renderings entspricht. Dann geht es ohne große Überraschungen weiter wie gehabt. Übrigens: Das Template ‘zf-deals/emails/say-hi-to-user’ habe ich zuvor in der config-Datei des Moduls, in der die E-Mail versendet wird, definiert, handelt es sich wie üblich bei “zf-deals/emails/say-hi-to-user” doch wieder nur um einen symbolischen Namen für ein Template und nicht etwa um eine Pfadangabe:

```
1  <?php
2  // [...]
3  'view_manager' => array(
4      'template_map' => array(
5          'helloworld/emails/say-hi-to-user'
6              => __DIR__ . '/../view/emails/say-hi-to-user.phtml'
7          // [...]
8      )
9  )
10 // [...]
```

Listing 23.5

23.4 E-Mails mit Anhängen versenden

Ähnlich simpel gestaltet sich der Versand einer E-Mail mit Anhang durch Nutzung eines zusätzlichen “Mime Parts”. Im folgenden Beispiel wird exemplarisch eine Textdatei als Anhang versendet:

```
1  <?php
2  $view = $this->getServiceLocator()->get('ViewManager')->getView();
3
4  $viewModel = new \Zend\View\Model\ViewModel(
5      array('name' => 'Michael')
6  );
7
8  $viewModel->setOption('has_parent', true);
9  $viewModel->setTemplate('zf-deals/emails/say-hi-to-user');
10
11 $textPart = new \Zend\Mime\Part('Hi, Michael!');
12 $textPart->type = "text/plain";
13
14 $file = __DIR__ . '/attachment.txt';
15 $attachment = new \Zend\Mime\Part(file_get_contents($file));
16 $attachment->filename = basename($file);
17 $attachment->disposition = \Zend\Mime\Mime::DISPOSITION_ATTACHMENT;
18 $attachment->encoding = \Zend\Mime\Mime::ENCODING_8BIT;
19
20 $body = new \Zend\Mime\Message();
21 $body->setParts(array($textPart, $attachment));
22
23 $mail = new \Zend\Mail\Message();
24 $mail->setBody($body);
25 $mail->setFrom('sender@example.com', 'Sender name');
26 $mail->addTo('zf2buch@michael-romer.de', 'Michael Romer');
27 $mail->setSubject('Mail subject');
28
29 $transport = new \Zend\Mail\Transport\Sendmail();
30 $transport->send($mail);
```

Listing 23.6

24 Menüs & Co.

24.1 Navigationen erzeugen

Mit `Zend\Navigation` lassen sich Navigationselemente entwickeln, die über einen einzelnen, simplen Link oder Button hinausgehen. Also etwa komplexere Menüs, Linklisten oder ähnliches. Über sie wird letztlich in einer Anwendung das gesamte “Nutzer-Browsing” realisiert und daher kommt Framework-Unterstützung hier sehr gelegen.

Navigationen im Zend Framework 2 basieren, wie übrigens auch bereits in Version 1, auf zwei Kernelementen: “Pages” und “Containers”. Die Bezeichnung “Page” ist etwas irreführend, handelt es sich bei “Page” doch nicht um die eigentlich (Ziel-)Seite, sondern um einen “Zeiger” (“Pointer”) auf ebenjene. “Container” kommen zum Einsatz um “Zeiger” zu gruppieren und bringen die dafür notwendigen “Management-Funktionen” mit. Die Klasse `Zend\Navigation\Navigation` stellt die Implementierung eines solchen “Containers” dar, der zudem dazu in der Lage ist, auf Basis eines Konfigurations-Arrays die entsprechenden “Pages”-Objekte zu erzeugen. Schauen wir uns ein Beispiel an, dann wird es klarer. Ein leerer “Container” wird via

```
1 <?php
2 $container = new \Zend\Navigation\Navigation();
```

Listing 24.1

erzeugt. Zum jetzigen Zeitpunkt enthält er keine “Pages”, allerdings könnten wir sie etwa mit den Methoden `addPage()`, `setPages()`, usw. Stück für Stück hinzufügen, wenn wir die jeweils notwendigen “Pages” zuvor instanziiert hätten. Da dieses Vorgehen gerade bei komplexeren Navigationen aber sehr schnell sehr viel unnötige Schreibarbeit erfordert und die Übersichtlichkeit schwindet, empfiehlt es sich, die “Pages-Struktur” über ein Konfiguration-Array zu definieren. Angenommen, wir wollen eine Linkliste erzeugen, die auf jeden Fall schon einmal einen Link zu <http://www.zendframework2.de> enthält (-D), so würden wir Folgendes schreiben:

```
1 <?php
2 $container = new \Zend\Navigation\Navigation(array(
3     array(
4         'label' => 'ZF2 Book',
5         'id' => 'zf2book',
6         'uri' => 'http://www.zendframework2.de'
7     )
8 ));
```

Listing 24.2

Aus dem übergebenen Konfigurations-Array erzeugt `Navigation` nun die entsprechende “Page” für uns.

Zum aktuellen Zeitpunkt sehen wir allerdings ... noch nichts, denn analog zu den Webforms ist bei `Zend\Navigation` die Datenhaltung (das “Model”) strikt von dessen Darstellung getrennt. Für die Anzeige der Navigation kommen wie bei `Zend\Form` ebenfalls wieder einige “View Helper” zum Einsatz. Um den oben gezeigten Container nun also zu “rendern” reicht zunächst der folgende Aufruf innerhalb eines Templates, insofern der erstellte “Container” zuvor im Rahmen des “View Models” unter dem Key `container` bereitgestellt wurde:

```
1 <?php echo $this->navigation($this->container); ?>
```

Listing 24.3

Das Ergebnis ist wie folgt:

```
1 <ul class="navigation">
2     <li>
3         <a id="menu-zf2book" href="http://www.zendframework2.de">
4             ZF2 Book
5         </a>
6     </li>
7 </ul>
```

Listing 24.4

Soweit so gut. Grundsätzlich lassen sich “Pages” in zwei unterschiedlichen Typen erzeugen: Als `Uri` oder als `Mvc`. Den Typ `Uri` haben wir bereits oben eingesetzt. Er ermöglicht es, einen “Pointer” auf eine feste URL zu setzen und wird in der Regel insbesondere für externe Links eingesetzt. Der Typ `Mvc` hingegen erlaubt es, Zeiger auf Basis zuvor definierter Routen zu erzeugen. Das ist insbesondere dann hilfreich, wenn die jeweiligen URLs dynamische Bestandteile, etwa die Artikelnummer und den Namen eines Produktes in einem Online-Shop, haben. Wir können den “Container” von oben wie folgt um einen Link zurück zur Startseite der Anwendung erweitern:

```
1  <?php
2  $container = new \Zend\Navigation\Navigation(array(
3      array(
4          'label' => 'ZF2 Book',
5          'id' => 'zf2book',
6          'uri' => 'http://www.zendframework2.de'
7      ),
8      array(
9          'label' => 'Home',
10         'id' => 'home',
11         'route' => 'home'
12     )
13 ));
```

Listing 24.5

Bei Bedarf können über den Key `params` zusätzliche, für die Erzeugung der URL auf Basis der Route notwendige, dynamische Werte übergeben werden. Diese Konfiguration setzt allerdings voraus, dass es eine zuvor definierte Route “home” gibt:

```
1  <?php
2  // [...]
3  'router' => array(
4      'routes' => array(
5          'home' => array(
6              'type' => 'Zend\Mvc\Router\Http\Literal',
7              'options' => array(
8                  'route' => '/',
9                  'defaults' => array(
10                     'controller' => 'Application\Controller\Index',
11                     'action' => 'index',
12                 )
13             )
14         )
15     )
16 )
17 // [...]
```

Listing 24.6



Navigationselemente nur berechtigten Nutzern zeigen

Kommt bereits `Zend\Permissions\Acl` in einer Anwendung zum Einsatz, so können bestimmte “Pages” über den Key `resource` samt entsprechendem Wert situationsabhängig ein- bzw. ausgeblendet werden. Das Ganze lässt sich zusätzlich auch noch um den Key `privilege` erweitern und somit Zugangskontrollen noch feiner in den Navigationselementen berücksichtigen.

Nehmen wir einmal an, wir wollen eine umfassende Link-Sektion für externe Seiten aufbauen. Dazu legen wir alle einzelnen Links unter dem “Top-Level”-Navigationspunkt “Links” an. Wir können dieses Vorhaben wie folgt durch die entsprechende Schachtelung im Konfigurationsarray erreichen:

```
1  <?php
2  $container = new \Zend\Navigation\Navigation(array(
3      array(
4          'label' => 'Home',
5          'id' => 'home',
6          'route' => 'home'
7      ),
8      array(
9          'label' => 'Links',
10         'id' => 'links',
11         'uri' => '#',
12         'pages' => array(
13             array(
14                 'label' => 'ZF2 Download',
15                 'id' => 'zf2',
16                 'uri' => 'http://framework.zend.com'
17             ),
18             array(
19                 'label' => 'ZF2 Book',
20                 'id' => 'zf2book',
21                 'uri' => 'http://www.zendframework2.de'
22             )
23         )
24     )
25 ));
```

Listing 24.7

Das Ergebnis entspricht den Erwartungen:

```
1 <ul class="navigation">
2     <li>
3         <a id="menu-home" href="/">Startseite</a>
4     </li>
5     <li>
6         <a id="menu-links" href="#">Links</a>
7         <ul>
8             <li>
9                 <a id="menu-zf2" href="http://framework.zend.com">
10                     ZF2 Download
11                 </a>
12             </li>
13             <li>
14                 <a id="menu-zf2book" href="http://www.zendframework2.de">
15                     ZF2 Book
16                 </a>
17             </li>
18         </ul>
19     </li>
20 </ul>
```

Listing 24.8



Einen Container durchsuchen

Ein “Container” bringt eine ganze Reihe von Methoden mit, um eine oder mehrere “Pages” aufzufinden, etwa `findById()` oder `findAllByClass()`. Sie erweisen sich insbesondere als sehr hilfreich, wenn man ein fixes Konfigurationsarray einsetzt und kleinere Modifikationen situationsabhängig im Nachhinein über ebenjene Methoden durchführt.

Um die Darstellung des Menüs anzupassen, bieten sowohl die “Pages”, auch auch die entsprechenden “View Helper” einige Möglichkeiten. So lässt sich über

```
1 <?php echo $this->navigation($this->container)
2     ->menu()->setUIClass('menu'); ?>
```

Listing 24.9

die für das `ul`-Element des Menüs verwendete Klasse manipulieren oder via `setClass()` die CSS-Klasse einer “Page” setzten. Bei Bedarf kann man sogar ein eigenes Template für die Navigation

angeben, das über den “View Helper” `Partial` dann Anwendung für die Darstellung findet. So könnte man etwa Navigationen realisieren, die einen gänzlich anderen Aufbau haben und etwa nicht auf einer `ul` basieren.

24.2 Eine Anwendungsnavigation erzeugen

Schauen wir noch einmal detaillierter auf die CSS-Klasse einer “Page”, denn die ist in einer speziellen Situation sehr hilfreich: Nämlich immer dann, wenn ein Eintrag in der Navigation der URL entspricht, auf der man sich gerade befindet. In diesem Fall sorgt `Zend\Navigation` dafür, dass die “Page”, bzw. dessen HTML-Element ``, automatisch die CSS-Klasse `active` zugewiesen wird. Der Menüpunkt kann dann visuell andersartig, etwa in Fettschrift, dargestellt werden.

Damit das Ganze funktioniert, muss man die Sache aber etwas anders angehen. Zunächst einmal muss der “Container” nicht wie bisher direkt instanziiert werden

```
1 <?php
2 $container = new \Zend\Navigation\Navigation();
```

Listing 24.10

sondern stattdessen über die vom Framework mitgelieferte Factory erzeugt werden:

```
1 <?php
2 // [...]
3 service_manager => array(
4     'factories' => array(
5         'Navigation' => 'Zend\Navigation\Service\DefaultNavigationFactory',
6         // [...]
7     )
8 )
9 // [...]
```

Listing 24.11

Im Beispiel oben ist die entsprechende Factory unter dem Key `Navigation` im `ServiceManager` hinterlegt und kann so später einfach verwendet werden. Zusätzlich muss die `Navigation` wie gehabt selbst mit den `Pages` konfiguriert werden, was wir in diesem Fall ebenfalls in der `module.config.php` im Abschnitt `navigation` erledigen:


```

1  <?php
2  // [...]
3  'navigation' => array(
4      'default' => array(
5          array(
6              'label' => 'Home',
7              'id' => 'home',
8              'route' => 'home',
9          ),
10         array(
11             'label' => 'Links',
12             'id' => 'links',
13             'uri' => '#',
14             'pages' => array(
15                 array(
16                     'label' => 'ZF2 Download',
17                     'id' => 'zf2',
18                     'uri' => 'http://framework.zend.com'
19                 ),
20                 array(
21                     'label' => 'ZF2 Book',
22                     'id' => 'zf2book',
23                     'uri' => 'http://www.zendframework2.de'
24                 )
25             )
26         )
27     )
28 )
29 // [...]

```

Listing 24.12

Dafür können wir nun im Controller auf sämtliche Logik verzichten und in der View oder dem Layout unserer Wahl schlicht

```

1  <?php echo $this->navigation('Navigation')->menu(); ?>

```

Listing 24.13

aufrufen, um das gewünschte Ergebnis zu erzeugen. Der “View Helper” bedient sich selbst des ServiceManager und fordert den Service Navigation an, den “Container”, den er für alles Weitere benötigt. Die CSS-Klasse active wird nun entsprechend automatisch korrekt eingefügt. Selbstverständlich lassen sich bestimmte Seiten bei Bedarf nach wie vor auch manuell über die Methode setActive() als “aktiv” kennzeichnen.



Submenüs darstellen

Es lassen sich auch nur Teilbäume einer Navigation darstellen, etwa die sich unterhalb der aktiven “Page” befinden. Dazu verwendet man dann die `setOnlyActiveBranch()`-Methode des “View Helpers” Menu.

24.3 Breadcrumbs erzeugen

Hat man einmal eine “Anwendungsnavigation” konfiguriert, lässt sich sehr einfach auch eine sog. “Breadcrumb” erzeugen:

```
1 <?php echo $this->navigation('Navigation')->breadcrumbs(); ?>
```

Listing 24.14

Die “Breadcrumb” wird allerdings immer nur dann angezeigt, wenn die aktive “Page” nicht auf der obersten Ebene der Hierarchie ist.



Weitere Möglichkeiten

Zend\Navigation bringt darüber hinaus noch einen bunten Strauß zusätzlicher Möglichkeiten und Funktionen mit. Es lohnt sich, bei Navigationselementen zunächst immer die Dokumentation zu konsultieren und dort nach fertigen Lösungen für ein gegebenes Problem Ausschau zu halten, bevor man eine eigene Implementierung angeht.

25 Anwendungen durch Caching beschleunigen

25.1 Einleitung

Spätestens dann, wenn man eine gewisse Nutzerschaft um sich geschart hat, kommt man irgendwann an einen Punkt, an dem die eigene Anwendung beginnt träge zu reagieren. Im Folgenden sehen wir uns verschiedene Ansätze und Ideen an, wie man einer lahmen Anwendung wieder Beine macht. Alle Ansätze basieren auf der Idee, bestimmte Prozessschritte in der Verarbeitung eines Requests auszulassen und stattdessen einmalig berechnete und dann zwischengespeicherte Werte zu nutzen. Es wird sog. “Caching” betrieben.

25.2 Opcode-Cache

Eine einfache und zugleich sehr effektive Maßnahme ist der Einsatz von [APC](http://php.net/manual/de/book.apc.php)¹ als sog. “Opcode-Cache”. Konzeptbedingt wird normalerweise jedes Mal, wenn ein Skript zur Ausführung kommt, dieses neu kompiliert und entsprechend in den jeweils plattformspezifischen Maschinencode, sog. “Opcodes”, übersetzt. Da das Ergebnis der Übersetzung eines PHP-Skripts, das sich zwischendurch nicht geändert hat, in aller Regel identisch ist, bietet es sich an, die Übersetzung in Maschinencode nur einmalig auszuführen und das Ergebnis für die spätere Wiederverwendung zwischenzuspeichern. Alleine mit einem installieren und aktivieren “APC” sind Performancesteigerungen von 20-30% keine Seltenheit.



Alternative Implementierungen

Neben “APC” sind eine Reihe weiterer sog. “Opcode-Caches” (auch: “PHP Accelerators”) wie etwa [XCache](http://xcache.lighttpd.net/)² oder [ionCube](http://www.php-accelerator.co.uk/)³ verfügbar.

¹<http://php.net/manual/de/book.apc.php>

²<http://xcache.lighttpd.net/>

³<http://www.php-accelerator.co.uk/>

25.3 Zend\Cache

Ein weiterer Ansatz ist die Zwischenspeicherung von (Teil-)Ergebnissen, Objekten oder Response-Fragmenten. Während “APC” als “Opcode-Cache” wie oben beschreiben alle PHP-Skripte zu einem gewissen Maß automatisch beschleunigt, ist man als Anwendungsentwickler hier allerdings mehr gefordert, gilt es doch zu entscheiden, für welche Daten sich Caching lohnt, welche Maßnahmen später auch einen messbaren Effekt haben werden. Das Zend Framework 2 bringt mit Zend\Cache eine Komponente mit, die aber zumindest schon einmal die technischen Belange des Cachings maßgeblich vereinfacht. Zend\Cache erlaubt es, Daten zwischenzuspeichern, zuvor gespeicherte Daten wieder hervorzuholen und bei Bedarf meist sogar zunächst auf Existenz des Werts im Cache zu prüfen. Genau genommen sind dies allesamt die Aufgaben des Adapter, der jeweils für das “Storage” der Wahl die möglichen Funktionen bereitstellt. Als “Storage” können dabei verschiedene Systeme oder Ansätze zum Einsatz kommen, so wird u.a. neben “Memcached”, dem Dateisystem oder dem RAM des jeweiligen Systems auch der bereits im Rahmen des “Opcode-Cache” angesprochene “APC” als “Storage” unterstützt. Letzterer unterstützt eben nicht nur das “Opcode-Caching”, sondern speichert bei Bedarf auch beliebige andere Werte zwischen.

Ein “Cache” lässt sich am besten über die StorageFactory erzeugen:

```
1 <?php
2 $cache = \Zend\Cache\StorageFactory::factory(array(
3     'adapter' => 'apc',
4 ));
```

Listing 25.1

Ein Wert lässt sich dann einfach mit einem definierten Key hinterlegen

```
1 <?php
2 $cache->setItem('timestamp', time());
```

Listing 25.2

abrufen

```
1 <?php
2 if ($cache->hasItem('timestamp'))
3     var_dump($cache->getItem('timestamp'));
```

Listing 25.3

und löschen:

```
1 <?php
2 $cache->removeItem('timestamp');
```

Listing 25.4



Serialisierbarkeit

Damit sich ein Wert speichern lässt, muss er dem Kriterium der Serialisierbarkeit entsprechen. Kurz gesagt: Der Wert, etwa ein PHP-Objekt, muss sich für die Speicherung zunächst “zerlegen” und später auch konsistent wieder zusammenbauen lassen. Und das losgelöst von der seinerzeit präsenten Laufzeitumgebung. Versucht man Werte zu speichern, die sich nicht serialisieren bzw. deserialisieren lassen, wird diese Tatsache in der Regel direkt mit einer Exception quittiert, etwa mit “You cannot serialize or unserialize PDOStatement instances”.



Storage-Funktionsumfang

Je nach eingesetztem “Storage” hat die Cache-Instanz einen unterschiedlichen Funktionsumfang. Die Ausführungen in diesem Kapitel beziehen sich jeweils auf den “APC-Storage” und sind ggf. nicht ohne Anpassungen auf andere Systeme übertragbar. Übrigens: Mit Hilfe von “Storage Plugins” können “Adapter” mit Funktionalität nachgerüstet werden, die nicht “von Haus aus” verfügbar ist. Es besteht die Möglichkeit, sich vor der eigentlichen Aktion im “Storage” oder aber danach “einzuklinken”. Konsequenterweise nutzt der AbstractAdapter als Basis der konkreten “Adapter” den EventManager des Frameworks, der uns an so vielen Stellen bereits begegnet ist.

Über die “Adapter-Options” lässt sich unter anderem bestimmen, wie lange die Daten im Cache vorgehalten werden sollen. In diesem Beispiel werden die Daten für 10 Sekunden vorgehalten:

```
1 <?php
2 $cache = \Zend\Cache\StorageFactory::factory(array(
3     'adapter' => array(
4         'name' => 'apc',
5         'options' => array(
6             'ttl' => 10,
7         )
8     )
9 ));
```

Listing 25.5

Weitere Optionen, teils abhängig vom jeweiligen “Storage”, können auf diese Weise festgelegt werden.

Um die Arbeit mit dem Cache weiter zu vereinfachen, bringt das Zend Framework 2 noch einige maßgeschneiderte Caching-Lösungen, sog. “Caching-Patterns” mit. Bislang haben wir, wenn man denn so will, die “generische” Caching-Lösung eingesetzt, für die aber je nach Situation und Vorhaben noch ein mehr oder weniger großer Implementierungsaufwand auf den Anwendungsentwickler wartet. Daher kommen für die folgenden Situationen spezielle Lösungen zum Einsatz:

- **CaptureCache**: Speichern von “ganzen Seiten”, so dass beim nächsten Aufruf ohne weitere Verarbeitung direkt das Ergebnis zurückgeliefert werden kann. Auf diese Weise werden dynamische Inhalte temporär zu statischen Inhalten.
- **OutputCache**: Speichert Ausgaben zwischen einem definierten Start- und Endpunkt. Der OutputCache ist technisch gesehen ein Wrapper für die [Output-Control-Funktionen](http://php.net/manual/de/ref.outcontrol.php)⁴ des PHP Kerns.
- **ObjectCache**: Kommt zum Einsatz, um den Rückgabewert von Objekt-Methoden zu speichern. Technisch handelt es sich beim ObjectCache um einen Objekt-Wrapper.
- **ClassCache**: Kommt zum Einsatz, um den Rückgabewert von Klassen-Methoden zu speichern.
- **CallbackCache**: Analog zum ObjectCache und ClassCache speichert der CallbackCache das Ergebnis von Callbacks.

Damit gibt es im Grunde für alle Situationen, in denen Daten erzeugt oder geladen werden, eine spezielle Caching-Lösung und der Einsatz des “generischen Caches” wird nur noch in Ausnahmefällen notwendig sein. Schauen wir stellvertretend für die Speziallösungen einmal genauer auf den ObjectCache:

⁴<http://php.net/manual/de/ref.outcontrol.php>

```
1 <?php
2 $cachedVersion = \Zend\Cache\PatternFactory::factory('object', array(
3     'storage' => 'apc',
4     'object' => new \Zend\Version\Version()
5 ));
6
7 die($cachedVersion->getLatest());
```

Listing 25.6

Die Methode `getLatest()` von `Zend\Version\Version` holt via `file_get_contents` die aktuelle Versionsnummer des Zend Framework 2 von github. Weil hier ein entfernter Dienst aufgerufen wird, kann das Ergebnis schon mal einige Sekunden auf sich warten lassen. Daher ist dieser Methodenaufruf ein guter Kandidat für Caching. Führt der erste Aufruf von `getLatest()` auf dem “Wrapper-Objekt” `$cachedVersion` intern noch zum Aufruf von `file_get_contents`, so partizipieren alle Folgenden bereits vom Caching. Der Zugriff auf github entfällt, das zuvor bereits ermittelte Ergebnis wird direkt ausgegeben.

Über die Optionen `object_cache_methods` bzw. `object_non_cache_methods` der `PatternFactory` lassen sich übrigens gezielt auch nur einzelne Methoden für das Caching vorsehen, bzw. ganz vom Caching ausschließen.



CaptureCache-Alternative: Varnish

Will man “ganze Seiten” cachen, empfiehlt sich statt des CaptureCache eigentlich der Einsatz eines “Reverse Proxy” wie [Varnish](https://www.varnish-cache.org/)⁵. Er ist noch einmal ein gutes Stück performanter und hilft dabei, die eigene Anwendung von “Caching-Code” freizuhalten, der die Anwendung komplexer, fehleranfällig und häufig auch abhängig von externen Systemen oder etwa Schreibrechten auf bestimmten Verzeichnissen macht. “Varnish” ist simpel, effektiv und effizient.

25.4 Performance-Flaschenhälse identifizieren

Manchmal ist es gar nicht so leicht, nur beim Blick auf den Code zu beurteilen, ob Caching an einer bestimmten Stelle sinnvoll und hilfreich ist oder nicht. Der Einsatz sog. “Profiling-Tools” kann hier eine enorme Hilfe sein. So bekommt man über [Xdebug](http://xdebug.org/)⁶, [xhprof](http://pecl.php.net/package/xhprof)⁷ oder auch der kommerziellen,

⁵<https://www.varnish-cache.org/>

⁶<http://xdebug.org/>

⁷<http://pecl.php.net/package/xhprof>

aber sehr zu empfehlenden SaaS-Lösung [New Relic](http://newrelic.com)⁸ Einblicke in die Skriptverarbeitung, die viele ungeahnte Optimierungen möglich machen. Es lohnt sich also sehr, sich mit diesen Tools intensiver zu beschäftigen.

⁸<http://newrelic.com>

26 Entwicklertagebuch (Praxisteil)

26.1 Einführung

Während wir im ersten Teil des Buchs vornehmlich die theoretischen Konzepte und die individuellen Komponenten des Zend Framework 2 betrachtet haben, geht es nun in ein durchgängiges Praxisbeispiel, im dem insbesondere auch die Details der Entwicklung einer Anwendung auf Basis des Zend Framework 2 Beachtung finden - denn die sind es ja häufig, die es am Ende ausmachen.

Für die Entwicklung der Anwendung werden wir zudem die Methode [Scrum](http://de.wikipedia.org/wiki/Scrum)¹ einsetzen - zumindest soweit es im diesem Rahmen möglich ist. Ich selbst arbeite jetzt bereits seit einigen Jahren mit agilen Methoden und seit 2008 speziell mit Scrum. Ich halte Scrum für wirklich gut und habe in vielen Projekten durchweg sehr positive Erfahrungen gemacht. Ich kann diese Methode also für ernsthafte Projekte wirklich nur dringend empfehlen. Denn: Technische Exzellenz, bei der uns das Zend Framework 2 maßgeblich unterstützt, reicht alleine nicht für den Projekterfolg aus. Nicht weniger wichtig ist die Art und Weise, wie sich die Teammitglieder organisieren. Und Scrum ist genau dafür ein sehr hilfreiches Framework.

Wir starten mit dem sog. "Envisioning", dass dabei hilft, die Vorstellung für das Produkt, auf das man später dann Sprint für Sprint, Iteration für Iteration hinarbeitet, zu schärfen. Das "Envisioning" kann je nach Situation sehr unterschiedlich ausgestaltet werden und das Scrum Framework macht auch keinerlei Aussagen dazu, wie diese Projektphase auszusehen hat. Ich möchte das "Envisioning" im Rahmen dieses Buch dazu nutzen, die Idee hinter der zu entwickelnden Demo-Anwendung zu erläutern. Danach arbeiten wir Iteration für Iteration an der Realisierung der gewünschten Funktionalität.

26.2 Envisioning

Im Rahmen dieses Praxisbeispiels möchte ich gerne die Basisfunktion von "ZfDeals" entwickeln. "ZfDeals" ist eine Software zum Verkauf von Produkten zu Sonderpreisen, sog. "Deals". Wir werden "ZfDeals" so programmieren, dass wir später die eigentliche Kernfunktionalität im Rahmen eines Moduls anderen Zend Framework 2 Anwendern zur Verfügung stellen können. Gemeinsam mit dem wiederverwendbaren Modul soll eine Demo-Anwendung erstellt werden, so dass man das Modul direkt in Aktion sehen kann.

¹<http://de.wikipedia.org/wiki/Scrum>

26.3 Sprint 1 - Code-Repository, Entwicklungsumgebung, initiale Codebase

Im ersten Sprint geht es mir darum, mich in die Situation zu versetzen, mit der Entwicklung der Anwendung beginnen zu können.

Git-Repository mit Basiscode erstellen

Dazu benötige ich zunächst einmal ein System zur Verwaltung des Codes - schließlich will ich jederzeit zu alten Versionen meiner Dateien zurück, regelmäßig Branches erstellen und überhaupt den Code an einem sicheren Ort wissen. Meine Wahl fällt auf Git lokal und zusätzlich auch bei [Github](https://github.com/)². Dort verfüge ich bereits über einen Account, andernfalls hätte ich mir in wenigen Minuten einen Account anlegen können. Github ist kostenlos für öffentliche Projekte, also für diesen Fall prima geeignet.

Git ist auf meinem System bereits vorinstalliert, so dass ich mit dem Aufruf

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

in einem Verzeichnis meiner Wahl die ZendSkeletonApplication klonen und als Ausgangspunkt für meine Anwendung nutzen kann.

Der Aufruf von Composer sorgt nun dafür, dass auch das Zend Framework 2 im Unterverzeichnis vendor bereitgestellt wird:

```
1 $ cd ZfDealsApp  
2 $ php composer.phar install
```

Jetzt erstelle ich mir noch ein Repository auf github, vornehmlich als eleganten Weg, um die Daten remote zu spiegeln und gleichzeitig auf einfache Weise für dich zur Verfügung stellen zu können. Zunächst entferne ich die Referenz auf das ursprüngliche Remote "origin" via

```
1 $ git remote rm origin
```

um sie dann auf das neu erzeugte Repository zu setzen:

```
1 $ git remote add origin https://github.com/michael-romer/ZfDealsApp.git
```

Ein abschließendes

²<https://github.com/>

```
1 $ git push -u origin master
```

schreibt nun die Daten in das entsprechende [Repository bei github](#)³.

Lokale Entwicklungsumgebung einrichten

Um die Anwendung auf meinem System laufen lassen zu können, erzeuge ich mir eine Virtuelle Maschine, anstatt die Dienste lokal zu installieren. Dazu verwende ich eine Codebibliothek, die ich schon des öfteren im Einsatz habe und die ich ebenfalls in meinem github-Account [in einem eigenen Repository](#)⁴ zur Verfügung stelle. Um die Bibliothek zu verwenden, greifen wir wieder auf Composer zurück und erweitern die `composer.json` wie folgt:

```
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        "zfb/zfb-vm": "dev-master"
14    }
15 }
```

Neu ist hier das `"zfb/zfb-vm": "dev-master"`. Der Aufruf

```
1 $ php composer.phar update
```

sorgt dafür, dass die Bibliothek heruntergeladen wird. Jetzt noch die Datei `/vendor/zfb/zfb-vm/Vagrantfile.dist` nach `/Vagrantfile` kopieren (also im Zuge des Kopierens umbenennen) und die folgenden Tools lokal installieren:

- [Virtual Box](#)⁵

³<https://github.com/michael-romer/ZfDealsApp>

⁴<https://github.com/michael-romer/zfb2-vm>

⁵<https://www.virtualbox.org/wiki/Downloads>

- [Ruby](#)⁶
- [Vagrant](#)⁷

Zugegeben, ein wenig Aufwand, aber eine einmalige Sache, denn nun kannst du dir für beliebig viele Projekte auf Kommando eine dedizierte, vollständige Laufzeitumgebung erzeugen. Und die Installation der Dienste direkt auf dem eigenen System wäre vermutlich auch nicht viel weniger aufwändig gewesen.

Jetzt noch die beiden folgenden Kommandos ausführen und in die Mittagspause gehen:

```
1 $ vagrant box add precise64 http://files.vagrantup.com/precise64.box
2 $ vagrant up
```

Danach kannst du, wenn alles glatt gelaufen ist, im Browser die URL `localhost:8080` aufrufen und wirst freundlich begrüßt. Mit Hilfe des Kommandos

```
1 $ vagrant ssh
```

oder [Putty](#), falls du Windows einsetzt⁸, kommst du einfach auf die Shell der VM (und über den Befehl `exit` wie gewohnt auch wieder heraus). Dort entspricht der Inhalt des `/vagrant`-Verzeichnisses dem des Projektverzeichnisses auf deinem lokalen System. Es ist zwischen den beiden Systemen “geshared”. Der Port 8080 auf deinem lokalen System wurde automatisch so eingerichtet, dass der auf Port 80 der virtuellen Maschine weiterleitet.

Zum Feierabend die VM via

```
1 $ vagrant suspend
```

im Projektverzeichnis des lokalen Systems in den Ruhezustand versetzen und am nächsten Tag via

```
1 $ vagrant resume
```

wieder aufwecken. Wird die VM eine Weile nicht benötigt, kann sie via

```
1 $ vagrant halt
```

gestoppt und mit

⁶<http://www.ruby-lang.org/de/>

⁷<http://vagrantup.com/>

⁸<http://vagrantup.com/v1/docs/getting-started/ssh.html>

```
1 $ vagrant up
```

wieder gestartet werden.

Natürlich ist eine VM nicht notwendig, um mit dem Zend Framework 2 Anwendungen zu entwickeln. Du kannst auch händisch die einzelnen Dienste auf deinem System installieren oder dich einem Paket wie [XAMPP](#)⁹ bedienen. Hauptsache es steht irgendwann dann eine Laufzeitumgebung mit PHP 5.3.3 (oder später) zur Verfügung und das Document-Root des Webservers ist so konfiguriert, dass es auf das Unterverzeichnis `public` des Projektverzeichnisses zeigt. Denn dort liegt die `index.php`, der zentrale Einstiegspunkt in die Anwendung.

Solange du also irgendwann so wie ich

```
1 "Welcome to Zend Framework 2"
```

auf dem Bildschirm siehst, hast auch du den ersten Sprint mit Bravour gemeistert!

26.4 Sprint 2 - Modul anlegen, Produkte eintragen



Quellcode-Download

Den Quellcode nach Abschluss dieses Sprints findest du zum Download im Tag [“Sprint2”](#) bei [github](#)¹⁰.

User Stories

Während das Erstellen der initialen Codebase und das Einrichten der Arbeitsumgebung als technische Vorarbeit verstanden werden kann, handelt es sich bei der Anforderung, neue Produkte im System hinterlegen zu können, um die erste “fachliche” Aufgabe. Im Rahmen von Scrum werden Anforderung in Form sog. “User Stories” formuliert:

“Eine User Story (‘Anwendererzählung’) ist eine in Alltagssprache formulierte Software-Anforderung. Sie ist bewusst kurz gehalten und umfasst in der Regel nicht mehr als zwei Sätze.” (Wikipedia)

⁹<http://www.apachefriends.org/de/xampp.html>

¹⁰<https://github.com/michael-romer/ZfDealsApp/tree/Sprint2>

Üblicherweise folgen User Stories einem bestimmten Muster. Meine Lieblingsvariante ist:

“Damit ich [Nutzen], möchte ich als [Rolle], dass [Wunsch/Funktion].”

In diesem Fall sieht die User Story wie folgt aus:

“Damit ich später ein Produkt als Deal anbieten kann, möchte ich als Händler, dass ich mein Inventar erfassen kann.”

Das klingt soweit ja auch ganz sinnvoll. Zu einer User Story gibt es dann gewöhnlich noch eine Reihe sog. “Akzeptanzkriterien”, die die eigentliche Anforderung genauer spezifizieren:

- Es lässt sich pro Produkt eine Produkt-ID (eindeutig), eine Produktbezeichnung und der Lagerbestand erfassen.
- Die Erfassung ist über ein Formular möglich.

ZF2-Modul anlegen

Dann lege ich mal los. Zunächst benötige ich ein neues ZF2-Modul, in dem die eigentliche Funktionalität von ZfDeals implementiert wird. Dazu lege ich manuell die folgende Datenstruktur im Verzeichnis `module` an:

```
1  ZfDeals/  
2      Module.php  
3      config/  
4          module.config.php  
5      src/  
6          ZfDeals/  
7              Controller/  
8                  AdminController.php  
9      view/  
10         zf-deals/  
11             admin/  
12                 index.phtml  
13         layout/  
14             admin.phtml
```



Zend\Tool

Zum aktuellen Zeitpunkt gibt es leider noch kein Zend\Tool, das einem diese Arbeit abnehmen könnte. Das aus Version 1 bekannte und hilfreiche Werkzeug ist erst für ein späteres Minor-Release des Frameworks geplant.

Die `Module.php` statte ich zunächst nur mit dem Notwendigsten aus, also mit Informationen zum Autoloading und einem Verweis auf die `module.config.php`, der Konfigurationsdatei des ZfDeals-Moduls:

```

1  <?php
2  namespace ZfDeals;
3
4  class Module
5  {
6      public function getConfig()
7      {
8          return include __DIR__ . '/config/module.config.php';
9      }
10
11     public function getAutoloaderConfig()
12     {
13         return array(
14             'Zend\Loader\StandardAutoloader' => array(
15                 'namespaces' => array(
16                     __NAMESPACE__
17                         => __DIR__ . '/src/' . __NAMESPACE__,
18                 ),
19             ),
20         );
21     }
22 }
```

Listing 26.1

Den `AdminController` erzeuge ich zunächst auch “nackt”. Er basiert auf dem `AbstractActionController` des Frameworks, der es ermöglicht, mit einzelnen Actions innerhalb von Controllern zu arbeiten:

```

1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6
7  class AdminController extends AbstractActionController
8  {
9      public function indexAction()
10     {
11         return new ViewModel();
12     }
13 }

```

Listing 26.2

Die `module.config.php` statte ich ebenfalls zunächst nur mit dem Notwendigsten aus: Eine Route für die “Startseite” der Administration

```

1  <?php
2  return array(
3      'router' => array(
4          'routes' => array(
5              'zf-deals\admin\home' => array(
6                  'type' => 'Zend\Mvc\Router\Http\Literal',
7                  'options' => array(
8                      'route' => '/deals/admin',
9                      'defaults' => array(
10                         'controller'
11                             => 'ZfDeals\Controller\Admin',
12                         'action'
13                             => 'index',
14                     ),
15                 ),
16             ),
17         ),
18     ),
19     // [...]
20 )

```

Listing 26.3

den `AdminController` als “invokable”


```

1  <?php
2  // [...]
3  'controllers' => array(
4      'invokables' => array(
5          'ZfDeals\Controller\Admin'
6              => 'ZfDeals\Controller\AdminController'
7      ),
8  ),
9  // [...]

```

Listing 26.4

und die Konfiguration des admin-Layouts. In der Module-Klasse Sorge ich nun noch dafür, dass immer dann, wenn der AdminController in Aktion versetzt wird, auch das passende Layout verwendet wird:

```

1  <?php
2  // [...]
3  public function init(\Zend\ModuleManager\ModuleManager $moduleManager)
4  {
5      $sharedEvents = $moduleManager
6          ->getEventManager()->getSharedManager();
7      $sharedEvents->attach(
8          'ZfDeals\Controller\AdminController',
9          'dispatch',
10         function($e) {
11             $controller = $e->getTarget();
12             $controller->layout('zf-deals/layout/admin');
13         },
14         100
15     );
16 }
17 // [...]

```

Listing 26.5

Dazu registriere ich in der init-Methode eine Callback-Funktion für das dispatch-Event, das der ZfDeals\Controller\AdminController später einmal aussenden wird, wenn er denn an der Reihe ist. Weil der ZfDeals\Controller\AdminController samt seines eigenen EventManager zum Zeitpunkt der Listener-Registrierung noch nicht zur Verfügung steht, registriere ich die Funktion über den SharedEventManager. In der Funktion selbst wird dann nur noch das “Controller Plugin” layout verwendet. Das Argument des Aufrufs ist dabei der symbolische Name des admin-Layouts, wie in der module.config.php definiert:

```

1  <?php
2  // [...]
3  'view_manager' => array(
4      'template_map' => array(
5          'zf-deals/layout/admin' => __DIR__ . '/../view/layout/admin.phtml',
6      ),
7      'template_path_stack' => array(
8          __DIR__ . '/../view',
9      ),
10 ),
11 // [...]

```

Listing 26.6

Nicht vergessen, das neue Modul auch zu aktivieren. Dazu ist in der `application.config.php` im Verzeichnis `config` der folgende Abschnitt notwendig:

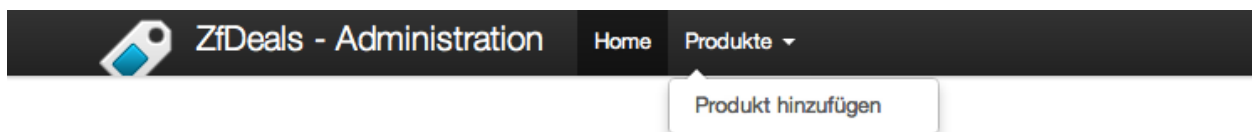
```

1  <?php
2  return array(
3      'modules' => array(
4          'Application',
5          'ZfDeals'
6      ),
7      // [...]
8  );

```

Listing 26.7

Wenn ich die `index`-Action im neuen Controller, sowie das entsprechende View-File erst einmal leer lasse (aber auf jeden Fall schon erstelle), kann ich die URL `/deals/admin` aufrufen und sehe immerhin bereits das Layout auf dem Bildschirm:



Basis-Layout von ZfDeals

Umgang mit Modul-Assets

Jetzt stellt sich mir grad die Frage, wie ich eigentlich mit den sog. “Assets” meines Moduls umgehe, also mit dem Images, CSS- und JS-Files, die zum Modul gehören. So will ich z.B. das “ZfDeals”-Logo

einbinden. Die Situation ist ja die Folgende: Will ich “Assets” bereitstellen, muss ich grundsätzlich dafür sorgen, dass sie im `public`-Verzeichnis der Anwendung verfügbar sind, nicht irgendwo vergraben in meinem Modul. Dort sind sie ohne Weiteres erst einmal nicht von Außen zugänglich. Zwar könnte ich über einen selbstentwickelten `file_get_contents()`-Mechanismus dafür sorgen, das “Assets”, die nicht direkt über eine URL zu erreichen sind, auf diese Weise verfügbar gemacht werden, nur ist das erstens nicht sehr elegant und zweitens auch ein “Performancefresser”, wenn man nicht auch gleich ein irgendwie geartetes Caching implementiert. Und das ist im Zweifelsfall auch nicht mal eben so gemacht. Das Framework bringt hier keine Unterstützung “out-of-the-Box”. Ich sehe demnach für mich für den Moment diese drei Möglichkeiten:

- Ich lege die Assets des Moduls im `public`-Verzeichnis der Anwendung ab und dort in einem Unterverzeichnis, das nach dem Modul benannt ist. Das würde aber bedeuten, dass, insofern das `ZfDeals`-Modul später auch eigenständig einsetzbar sein soll, bei der Installation immer auch die entsprechenden Dateien in das `public`-Verzeichnis der Anwendung kopiert werden müssen. Das müsste dann entweder manuell oder im Rahmen eines (Build-)Skripts geschehen.
- Ich lasse die Assets ausschließlich im `public`-Verzeichnis des Moduls und setze im `public`-Verzeichnis der Anwendung nur entsprechende Symlinks. Dieses Vorgehen wäre der ersten Option recht ähnlich, aber immerhin bleiben die Assets des Moduls innerhalb des Moduls.
- Ich verwende einen “Asset”-Manager, der sich um die Bereitstellung der “Assets” aus meinem Model heraus kümmert.

Ich favorisiere zwar den zuletzt genannten Weg, auch, weil er viele weitere Vorteile hat, wähle für den Moment aber die pragmatische Lösung, die Dateien zunächst einmal im `public`-Verzeichnis der Anwendung abzulegen. Einen “Asset”-Manager werde ich etwas später noch hinzunehmen, sobald die Anwendung bereit dazu ist, in Form eines Moduls bereitgestellt zu werden. Aber alles zu seiner Zeit.

Formular für die Erfassung von Produkten

Jetzt lege ich das Formular an, über das sich neue Produkte im System erfassen lassen. Ich mache es hier jetzt so, dass ich das Formular mit einem Fieldset bestücke, dass alle “produktbezogenen” Elemente kapselt:

```
1  <?php
2  namespace ZfDeals\Form;
3
4  use Zend\Form\Form;
5
6  class ProductAdd extends Form
7  {
8      public function __construct()
9      {
10         parent::__construct('login');
11         $this->setAttribute('action', '/deals/admin/product/add');
12         $this->setAttribute('method', 'post');
13
14         $this->add(array(
15             'type' => 'ZfDeals\Form\ProductFieldset',
16             'options' => array(
17                 'use_as_base_fieldset' => true
18             )
19         ));
20
21         $this->add(array(
22             'name' => 'submit',
23             'attributes' => array(
24                 'type' => 'submit',
25                 'value' => 'Hinzufügen'
26             ),
27         ));
28     }
29 }
```

Listing 26.8

Das Formular ProductAdd hält demnach selbst nur den “Hinzufügen”-Button, die eigentlichen Eingabefelder kommen über das ProductFieldset in die Webform, in dem ich auch gleich die notwendigen Validatoren hinterlege:

```
1  <?php
2  namespace ZfDeals\Form;
3
4  use Zend\Form\Fieldset;
5  use Zend\InputFilter\InputFilterInterface;
6  use Zend\InputFilter\InputFilterProviderInterface;
7
8  class ProductFieldset extends Fieldset
9      implements InputFilterProviderInterface
10 {
11     public function __construct()
12     {
13         parent::__construct('product');
14
15         $this->add(array(
16             'name' => 'id',
17             'attributes' => array(
18                 'type' => 'text',
19             ),
20             'options' => array(
21                 'label' => 'Produkt-ID:',
22             )
23         ));
24
25         $this->add(array(
26             'name' => 'name',
27             'attributes' => array(
28                 'type' => 'text',
29             ),
30             'options' => array(
31                 'label' => 'Produktbezeichnung:',
32             )
33         ));
34
35         $this->add(array(
36             'name' => 'stock',
37             'attributes' => array(
38                 'type' => 'number',
39             ),
40             'options' => array(
41                 'label' => '# Bestand:'
```

```
43         ),
44     ));
45 }
46
47 public function getInputFilterSpecification()
48 {
49     return array(
50         'id' => array (
51             'required' => true,
52             'filters' => array(
53                 array(
54                     'name' => 'StringTrim'
55                 )
56             ),
57             'validators' => array(
58                 array(
59                     'name' => 'NotEmpty',
60                     'options' => array(
61                         'message' =>
62                             "Bitte geben Sie die Produkt-ID an."
63                     )
64                 )
65             )
66         ),
67         'name' => array (
68             'required' => true,
69             'filters' => array(
70                 array(
71                     'name' => 'StringTrim'
72                 )
73             ),
74             'validators' => array(
75                 array(
76                     'name' => 'NotEmpty',
77                     'options' => array(
78                         'message' =>
79                             "Bitte geben Sie eine Produktbezeichnung an."
80                     ),
81                 )
82             )
83         ),
84         'stock' => array (
```

```

85         'required' => true,
86         'filters' => array(
87             array(
88                 'name' => 'StringTrim'
89             )
90         ),
91         'validators' => array(
92             array(
93                 'name' => 'NotEmpty',
94                 'options' => array(
95                     'message' =>
96                     "Bitte geben Sie die Lagerbestand an."
97                 )
98             ),
99             array(
100                 'name' => 'Digits',
101                 'options' => array(
102                     'message' =>
103                     "Bitte geben Sie einen ganzzahligen Wert an."
104                 )
105             ),
106             array(
107                 'name' => 'GreaterThan',
108                 'options' => array(
109                     'min' => 0,
110                     'message' =>
111                     "Bitte geben Sie Wert >= 0 an."
112                 )
113             )
114         )
115     );
116 }
117 }
118 }

```

Listing 26.9

So kann ich später die gleichen Feld-Definitionen im Rahmen eines anderen Formulars wiederverwenden, über das z.B. die Daten des Produkts bearbeitet werden können. Zwar ist Letzteres bislang keine konkrete Anforderung und deshalb sollte ich mich streng genommen jetzt gedanklich noch gar nicht damit beschäftigen, allerdings ist der Mehraufwand für ein Fieldset sehr gering und die Wahrscheinlichkeit, dass sich zum “add” auch irgendwann noch ein “edit” für das Product gesellt, aus Erfahrung hoch. Daher lohnt es sich, diesen einen Schritt direkt weiterzugehen.

Formulardarstellung

Eine zusätzliche Route & Action sorgen nun für die Anzeige und Verarbeitung der Form:

```

1  <?php
2  return array(
3      'router' => array(
4          'routes' => array(
5              'zf-deals\admin\home' => array(
6                  'type' => 'Zend\Mvc\Router\Http\Literal',
7                  'options' => array(
8                      'route' => '/deals/admin',
9                      'defaults' => array(
10                         'controller'
11                             => 'ZfDeals\Controller\Admin',
12                         'action'
13                             => 'index',
14                     ),
15                 ),
16             ),
17             'zf-deals\admin\product\add' => array(
18                 'type' => 'Zend\Mvc\Router\Http\Literal',
19                 'options' => array(
20                     'route' => '/deals/admin/product/add',
21                     'defaults' => array(
22                         'controller'
23                             => 'ZfDeals\Controller\Admin',
24                         'action'
25                             => 'add-product',
26                     ),
27                 ),
28             ),
29         ),
30     ),
31     // [...]
32 )

```

Listing 26.10

Die Action gestaltet sich zu diesem Zeitpunkt wie folgt:


```
1  <?php
2  // [...]
3  public function addProductAction()
4  {
5      $form = new \ZfDeals\Form\ProductAdd();
6
7      if ($this->getRequest()->isPost()) {
8          $form->setData($this->getRequest()->getPost());
9
10         if ($form->isValid()) {
11             // todo
12         } else {
13             return new ViewModel(
14                 array(
15                     'form' => $form
16                 )
17             );
18         }
19     } else {
20         return new ViewModel(
21             array(
22                 'form' => $form
23             )
24         );
25     }
26 }
27 // [...]
```

Listing 26.11

In dessen View Sorge ich noch dafür, dass das Formular auch angezeigt wird:

```
1  <?php
2  $this->form->prepare();
3  echo $this->form()->openTag($this->form);
4  echo $this->formRow($this->form->get('product')->get('id'));
5  echo $this->formRow($this->form->get('product')->get('name'));
6  echo $this->formRow($this->form->get('product')->get('stock'));
7  echo $this->formSubmit($this->form->get('submit'));
8  echo $this->form()->closeTag();
```

Listing 26.12

Wenn ich nun die URL `/deals/admin/product/add` aufrufe, sehe ich die Webform. Allerdings sieht sie so noch nicht wirklich gut aus und greift auch keine visuellen Komponenten von “[Twitter Bootstrap](http://twitter.github.com/bootstrap/)¹¹” auf, das ich gerne auch intensiv nutzen möchte. Damit ich nun nicht selbst die notwendigen CSS-Klassen hinzufügen und das durch die “Form View Helper” erzeugte Markup manipulieren muss, installiere ich zusätzlich noch das Modul “[DluTwBootstrap](https://bitbucket.org/dlu/dlutwbootstrap/overview)¹²”, das diese Aufgabe für mich übernimmt. Wie gewohnt mache ich das wieder über Composer. Der Eintrag für die `composer.json` lautet:

```
1 "dlu/dlutwbootstrap": "dev-master"
```

Ein schnelles

```
1 $ php composer.phar update
```

und schon ist das Modul installiert (das Aktivieren über die `application.config.php` nicht vergessen!). Es registriert vornehmlich eine ganze Reihe zusätzlicher “View Helper”, die von der Benennung her alle den standardmäßigen “Form View Helper” des Frameworks ähneln, allerdings dafür sorgen, dass die von “Twitter Bootstrap” bekannte Formular-Optik Anwendung findet:

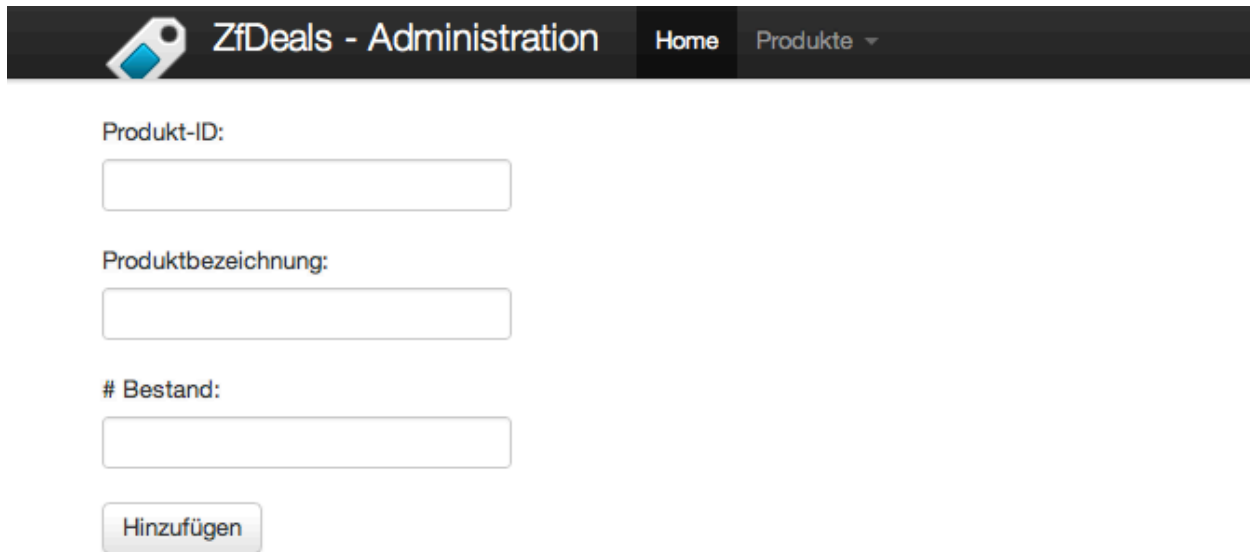
```
1 <?php
2 $this->form->prepare();
3 echo $this->form()->openTag($this->form);
4 echo $this->formRowTwb($this->form->get('product')->get('id'));
5 echo $this->formRowTwb($this->form->get('product')->get('name'));
6 echo $this->formRowTwb($this->form->get('product')->get('stock'));
7 echo $this->formSubmitTwb($this->form->get('submit'));
8 echo $this->form()->closeTag();
```

Listing 26.13

Und so stellt sich das Ergebnis dar:

¹¹<http://twitter.github.com/bootstrap/>

¹²<https://bitbucket.org/dlu/dlutwbootstrap/overview>



The screenshot shows the 'ZfDeals - Administration' web interface. At the top, there is a dark navigation bar with a logo on the left and two links, 'Home' and 'Produkte', on the right. Below the navigation bar, the main content area contains a form with three input fields: 'Produkt-ID:', 'Produktbezeichnung:', and '# Bestand:'. Each field is represented by a text label followed by an empty rectangular input box. Below these fields is a button labeled 'Hinzufügen'.

ZfDeals - Produkt hinzufügen

Unit-Tests für das Formular

Bevor ich mich damit beschäftige, die Daten aus dem Formular über den Weg einer Entity in die Datenbank zu schreiben, lege ich nun zunächst Unit Tests für die `ProductAdd`-Form an, damit ich sicher gehen kann, dass ich sie auch korrekt konfiguriert habe.

Dazu erstelle ich im Root der Anwendung ein neues Verzeichnis `tests`, im dem ich die `phpunit.xml` erstelle, über die ich festlege, welche Test-Unterverzeichnisse ich beim Ausführen der Tests später automatisch berücksichtigen will:

```
1 <phpunit bootstrap="./bootstrap.php">
2     <testsuites>
3         <testsuite name="AllTests">
4             <directory>./ZfDealsTest/FormTest</directory>
5         </testsuite>
6     </testsuites>
7 </phpunit>
```

Zudem brauche ich auch für die Ausführung der Tests noch die `bootstrap.php`, die analog zur “normalen” Ausführung der Anwendung dafür sorgt, dass die Anwendung samt allen Modulen initialisiert ist, bevor die eigentlichen Tests ausgeführt werden:

```
1  <?php
2  use Zend\Loader\StandardAutoloader;
3
4  chdir(dirname(__DIR__));
5
6  include 'init_autoloader.php';
7
8  $loader = new StandardAutoloader();
9  $loader->registerNamespace('ZfDealsTest', __DIR__ . '/ZfDealsTest');
10 $loader->register();
11
12 Zend\Mvc\Application::init(include 'config/application.config.php');
```

Listing 26.14

Zudem konfiguriere ich hier auch das Autoloading der Testklassen, die ich nun im Unterverzeichnis ZfDealsTest anlege. Die Tests der Webforms bringe ich dort im Verzeichnis FormTest unter und der ProductAddTest wird der erste konkrete Test:

```
1  <?php
2  namespace ZfDealsTest\FormTest;
3
4  use ZfDeals\Form\ProductAdd;
5
6  class ProductAddTest extends \PHPUnit_Framework_TestCase
7  {
8      private $form;
9      private $data;
10
11     public function setUp()
12     {
13         $this->form = new ProductAdd();
14         $this->data = array(
15             'product' => array(
16                 'id' => '',
17                 'name' => '',
18                 'stock' => ''
19             )
20         );
21     }
22
23     public function testEmptyValues()
24     {
```

```
25         $form = $this->form;
26         $data = $this->data;
27
28         $this->assertFalse($form->setData($data)->isValid());
29
30         $data['product']['id'] = 1;
31         $this->assertFalse($form->setData($data)->isValid());
32
33         $data['product']['name'] = 1;
34         $this->assertFalse($form->setData($data)->isValid());
35
36         $data['product']['stock'] = 1;
37         $this->assertTrue($form->setData($data)->isValid());
38     }
39
40     public function testStockElement()
41     {
42         $form = $this->form;
43         $data = $this->data;
44         $data['product']['id'] = 1;
45         $data['product']['name'] = 1;
46
47         $data['product']['stock'] = -1;
48         $this->assertFalse($form->setData($data)->isValid());
49
50         $data['product']['stock'] = "test";
51         $this->assertFalse($form->setData($data)->isValid());
52
53         $data['product']['stock'] = 12.3;
54         $this->assertFalse($form->setData($data)->isValid());
55
56         $data['product']['stock'] = 12;
57         $this->assertTrue($form->setData($data)->isValid());
58     }
59 }
```

Listing 26.15

Für die Tests erzeuge ich die Form und bestücke sie dann mit unterschiedlichen Testdaten. Ich prüfe jeweils, ob der Zustand der Form den Erwartungen entspricht und die Testdaten ein valides Ergebnis produzieren.

Product-Entity erstellen

Jetzt beginne ich damit, die “Business Domain” zu modellieren, in dem ich die Product-Entity erstelle:

```
1  <?php
2  namespace ZfDeals\Entity;
3
4  class Product
5  {
6      protected $id;
7      protected $name;
8      protected $stock;
9
10     public function setName($name)
11     {
12         $this->name = $name;
13     }
14
15     public function getName()
16     {
17         return $this->name;
18     }
19
20     public function setId($id)
21     {
22         $this->id = $id;
23     }
24
25     public function getId()
26     {
27         return $this->id;
28     }
29
30     public function setStock($stock)
31     {
32         $this->stock = $stock;
33     }
34
35     public function getStock()
36     {
37         return $this->stock;
```

```
38     }
39 }
```

Listing 26.16

Eine Product-Entity hält Informationen über seine Produkt-ID, seine Produktbezeichnung und den aktuellen Lagerbestand. Die Datei liegt wie gehabt im Verzeichnis `src/ZfDeals` des Moduls und dort im Unterverzeichnis `Entity`.

Persistenz für die Product-Entity

Passend dazu schreibe ich den “Mapper”, der die Attribute der Product-Entity auf eine Tabelle und deren Spalten in der Datenbank abbildet:

```
1  <?php
2
3  namespace ZfDeals\Mapper;
4
5  use ZfDeals\Entity\Product as ProductEntity;
6  use Zend\Stdlib\Hydrator\HydratorInterface;
7  use Zend\Db\TableGateway\TableGateway;
8  use Zend\Db\TableGateway\Feature\RowGatewayFeature;
9  use Zend\Db\Sql\Sql;
10 use Zend\Db\Sql\Insert;
11
12 class Product extends TableGateway
13 {
14     protected $tableName = 'product';
15     protected $idCol = 'id';
16     protected $entityPrototype = null;
17     protected $hydrator = null;
18
19     public function __construct($adapter)
20     {
21         parent::__construct($this->tableName,
22                             $adapter,
23                             new RowGatewayFeature($this->idCol)
24                         );
25
26         $this->entityPrototype = new ProductEntity();
27         $this->hydrator = new \Zend\Stdlib\Hydrator\Reflection;
28     }
```

```

29
30     public function insert($entity)
31     {
32         return parent::insert($this->hydrator->extract($entity));
33     }
34 }

```

Listing 26.17

Weil ich mich bei Strukturen in der Datenbank, insbesondere bei der Benennung von Spalten, an die Bezeichnungen aus der Product-Entity halte, fällt der Mapper sehr überschaubar aus. Über den `\Zend\Stdlib\Hydrator\Reflection-Hydrator` Sorge ich dafür, dass die Attribute der Entity beim `insert()` 1-zu-1 auf die Spalten der Tabelle “product” abgebildet werden.

Nun muss ich mich noch um den Datenbankadapter kümmern, der die Verbindung zur Datenbank herstellt. Ich konfiguriere ihn in der `module.config.php` des `ZfDeals`-Moduls:

```

1  <?php
2  // [...]
3  'service_manager' => array(
4      'factories' => array(
5          'Zend\Db\Adapter\Adapter' => function ($sm) {
6              $config = $sm->get('Config');
7              $dbParams = $config['dbParams'];
8
9              return new Zend\Db\Adapter\Adapter(array(
10                 'driver' => 'pdo',
11                 'dsn' =>
12                     'mysql:dbname=' . $dbParams['database']
13                     . ';host=' . $dbParams['hostname'],
14                 'database' => $dbParams['database'],
15                 'username' => $dbParams['username'],
16                 'password' => $dbParams['password'],
17                 'hostname' => $dbParams['hostname'],
18             ));
19         }
20     )
21 )
22 // [...]

```

Listing 26.18

Die eigentlichen Verbindungsdaten lege ich in der Datei `db.local.php` im Verzeichnis `/config/autoload` ab, die ich aber nicht in das Code-Repository einchecke, weil sie sensible Zugangsdaten enthält. Im

gleichen Atemzug erstelle ich aber auch die `db.local.php.dist`, die ich wiederum mit `einchecke`. Sie soll als “Blaupause” für diese Konfigurationsdatei dienen:

```
1 <?php
2 return array(
3     'dbParams' => array(
4         'database' => '',
5         'username' => '',
6         'password' => '',
7         'hostname' => '',
8     )
9 );
```

Listing 26.19

So ist klar, in welchem Format die Datenbank-Zugangsdaten auf dem jeweiligen Zielsystem bereitgestellt werden müssen.

In der `ServiceManager`-Konfiguration Sorge ich dafür, dass auch der `ZfDeals\Mapper\Product` über den `ServiceManager` verfügbar gemacht und die Abhängigkeit zum `Zend\Db\Adapter\Adapter` automatisch aufgelöst wird:

```
1 <?php
2 // [...]
3 'service_manager' => array(
4     'factories' => array(
5         'ZfDeals\Mapper\Product' => function ($sm) {
6             return new \ZfDeals\Mapper\Product(
7                 $sm->get('Zend\Db\Adapter\Adapter')
8             );
9         },
10    )
11 )
12 // [...]
```

Listing 26.20

Formularverarbeitung

Im Controller kann ich die Action jetzt dahingehend erweitern, dass ich die (validierten) Daten des Formulars auslese - bzw. direkt die passende Entity damit füllen lasse - und die Daten über den `ZfDeals\Mapper\Product` in die Datenbank schreibe:

```
1  <?php
2  // [...]
3  public function addProductAction()
4  {
5      $form = new \ZfDeals\Form\ProductAdd();
6
7      if ($this->getRequest()->isPost()) {
8          $form->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
9          $form->bind(new \ZfDeals\Entity\Product());
10         $form->setData($this->getRequest()->getPost());
11
12         if ($form->isValid()) {
13             $newEntity = $form->getData();
14
15             $mapper = $this->getServiceLocator()
16                 ->get('ZfDeals\Mapper\Product');
17
18             $mapper->insert($newEntity);
19             $form = new \ZfDeals\Form\ProductAdd();
20
21             return new ViewModel(
22                 array(
23                     'form' => $form,
24                     'success' => true
25                 )
26             );
27         } else {
28             return new ViewModel(
29                 array(
30                     'form' => $form
31                 )
32             );
33         }
34     } else {
35         return new ViewModel(
36             array(
37                 'form' => $form
38             )
39         );
40     }
41 }
42 // [...]
```

Listing 26.21

Aber natürlich erst, nachdem ich in der Datenbank “app” die entsprechende Tabelle erzeugt habe:

```
1 CREATE TABLE product(  
2     id varchar(255) NOT NULL,  
3     name varchar(255) NOT NULL,  
4     stock int(10) NOT NULL, PRIMARY KEY (id)  
5 );
```

Nun kann ich das Formular ausfüllen, abschicken und ein neuer Eintrag landet in der Datenbank. Im Erfolgsfall erzeuge ich eine Erfolgsmeldung, die oberhalb des Formulars angezeigt wird:

```
1 <?php if ($this->success) { ?>  
2 <div class="alert alert-success">Produkt hinzugefügt!</div>  
3 <?php } ?>  
4  
5 <?php  
6 $this->form->prepare();  
7 echo $this->form()->openTag($this->form);  
8 echo $this->formRowTwb($this->form->get('product')->get('id'));  
9 echo $this->formRowTwb($this->form->get('product')->get('name'));  
10 echo $this->formRowTwb($this->form->get('product')->get('stock'));  
11 echo $this->formSubmitTwb($this->form->get('submit'));  
12 echo $this->form()->closeTag();
```

Listing 26.22

Dependency Injection

Okay, soweit, so gut. Es gibt aber noch Einiges zu verbessern. An einigen Stellen kommt noch das Schlüsselwort `new` in meinem Code vor, werden Abhängigkeiten also explizit und vom Objekt selbst aufgelöst. Ich will von Anfang an möglichst darauf achten, solche Konstrukte zu vermeiden und wann immer möglich auf Dependency Injection zu setzen. Das wird mir das Testen erleichtern und meinen Code flexibler machen. Daher nehme ich an dieser Stelle noch ein paar Verbesserungen vor.

Die `AdminControllerFactory` soll von nun an dafür sorgen, dass die fertig konfigurierte Webform `ProductAdd` dem `AdminController` injiziert wird und auch um den `ZfDeals\Mapper\Product` muss sich der Controller aktiv demnächst nicht mehr kümmern:

```

1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class AdminControllerFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $ctr = new AdminController();
12         $form = new \ZfDeals\Form\ProductAdd();
13         $form->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
14         $form->bind(new \ZfDeals\Entity\Product());
15         $ctr->setProductAddForm($form);
16
17         $mapper = $serviceLocator->getServiceLocator()
18             ->get('ZfDeals\Mapper\Product');
19
20         $ctr->setProductMapper($mapper);
21         return $ctr;
22     }
23 }

```

Listing 26.23

In der `module.config.php` muss dann natürlich noch die passende Factory anstelle des “invokable” registriert werden:

```

1  <?php
2  // [...]
3  'controllers' => array(
4      'factories' => array(
5          'ZfDeals\Controller\Admin'
6              => 'ZfDeals\Controller\AdminControllerFactory'
7      )
8  )
9  // [...]

```

Listing 26.24

Unit-Tests für den Controller

Immer noch nicht perfekt, aber ein gutes Stück besser. Es fehlen mir aber auch noch die Unit Tests für den Controller, die ich aber eigentlich auch jetzt erst richtig erstellen kann, nachdem ich die zuvor genannten Verbesserungen vorgenommen habe.

Was will ich im Controller eigentlich genau testen? Weil der Controller selbst ja eigentlich gar nichts macht, sondern je nach Situation andere Objekte zur Arbeit auffordert, kann und sollte ich hier genau diesen Aspekt testen: Werden in den jeweiligen Situationen die richtigen Objekte angesprochen und die passenden Ergebnisse für das Rendering bereitgestellt? Im Grunde gibt es die folgenden Fälle:

1. Die Seite wird via GET aufgerufen: Das Formular, das dem Controller über die Factory bereitgestellt wurde, muss angezeigt werden.
2. Das Formular wurde abgeschickt (POST-Request), die Validierung schlägt fehl, das Formular muss wieder angezeigt werden.
3. Das Formular wurde abgeschickt, die Validierung ist erfolgreich, eine neue Entity, mit den im Formular eingegebenen Daten, wird dem "Mapper" zur Persistenz übergeben.

Und so sehen aktuell meine Unit Tests für den AdminController aus:

```
1  <?php
2  namespace ZfDeals\ControllerTest;
3
4  use ZfDeals\Controller\AdminController;
5  use Zend\Http\Request;
6  use Zend\Http\Response;
7  use Zend\Mvc\MvcEvent;
8  use Zend\Mvc\Router\RouteMatch;
9
10 class AdminControllerTest extends \PHPUnit_Framework_TestCase
11 {
12     private $controller;
13     private $request;
14     private $response;
15     private $routeMatch;
16     private $event;
17
18     public function setUp()
19     {
20         $this->controller = new AdminController();
21         $this->request = new Request();
22         $this->response = new Response();
```

```
23         $this->routeMatch = new RouteMatch(array('controller' => 'admin'));
24         $this->routeMatch->setParam('action', 'add-product');
25         $this->event = new MvcEvent();
26         $this->event->setRouteMatch($this->routeMatch);
27         $this->controller->setEvent($this->event);
28     }
29
30     public function testShowFormOnGetRequest()
31     {
32         $fakeForm = new \Zend\Form\Form('fakeForm');
33         $this->controller->setProductAddForm($fakeForm);
34         $this->request->setMethod('get');
35         $response = $this->controller->dispatch($this->request);
36         $viewModelValues = $response->getVariables();
37         $formReturned = $viewModelValues['form'];
38         $this->assertEquals($formReturned->getName(), $fakeForm->getName());
39     }
40
41     public function testShowFormOnValidationError()
42     {
43         $fakeForm = $this->getMock('Zend\Form\Form', array('isValid'));
44
45         $fakeForm->expects($this->once())
46             ->method('isValid')
47             ->will($this->returnValue(false));
48
49         $this->controller->setProductAddForm($fakeForm);
50         $this->request->setMethod('post');
51         $response = $this->controller->dispatch($this->request);
52         $viewModelValues = $response->getVariables();
53         $formReturned = $viewModelValues['form'];
54         $this->assertEquals($formReturned->getName(), $fakeForm->getName());
55     }
56
57     public function testCallMapperOnFormValidationSuccess()
58     {
59         $fakeForm = $this->getMock(
60             'Zend\Form\Form', array('isValid', 'getData')
61         );
62
63         $fakeForm->expects($this->once())
64             ->method('isValid')
```

```

65         ->will($this->returnValue(true));
66
67         $fakeForm->expects($this->once())
68             ->method('getData')
69             ->will($this->returnValue(new \stdClass()));
70
71         $fakeMapper = $this->getMock('ZfDeals\Mapper\Product',
72             array('insert'),
73             array(),
74             '',
75             false
76         );
77
78         $fakeMapper->expects($this->once())
79             ->method('insert')
80             ->will($this->returnValue(true));
81
82         $this->controller->setProductAddForm($fakeForm);
83         $this->controller->setProductMapper($fakeMapper);
84         $this->request->setMethod('post');
85         $response = $this->controller->dispatch($this->request);
86     }
87 }

```

Listing 26.25

Ein ganz guter Start, denke ich. Damit die Tests auch ausgeführt werden, habe ich die `phpunit.xml` noch entsprechend erweitert:

```

1 <phpunit bootstrap="./bootstrap.php">
2     <testsuites>
3         <testsuite name="AllTests">
4             <directory>./ZfDealsTest/FormTest</directory>
5             <directory>./ZfDealsTest/ControllerTest</directory>
6         </testsuite>
7     </testsuites>
8 </phpunit>

```

Doppelte Produkt-IDs vermeiden

Eine Sache, die mir jetzt so direkt noch auffällt: Was passiert eigentlich, wenn ich eine Produkt-ID eingebe, die bereits vergeben ist? Es kracht. Und zwar schon in der Datenbank, weil ich die

Spalte auf “unique” gesetzt habe (implizit; dadurch, dass es sich um den Primärschlüssel handelt). Ich entschieße mich dazu, diese Situation mit einem try/catch-Block an der entsprechenden Stelle zu behandeln:

```
1  <?php
2  public function addProductAction()
3  {
4      $form = $this->productAddForm;
5
6      if ($this->getRequest()->isPost()) {
7          $form->setData($this->getRequest()->getPost());
8
9          if ($form->isValid()) {
10             $model = new ViewModel(
11                 array(
12                     'form' => $form
13                 )
14             );
15
16             try {
17                 $this->productMapper->insert($form->getData());
18                 $model->setVariable('success', true);
19             } catch (\Exception $e) {
20                 $model->setVariable('insertError', true);
21             }
22
23             return $model;
24         } else {
25             return new ViewModel(
26                 array(
27                     'form' => $form
28                 )
29             );
30         }
31     } else {
32         return new ViewModel(
33             array(
34                 'form' => $form
35             )
36         );
37     }
38 }
```


Listing 26.26

Das View-File sieht nun so aus:

```
1  <?php if ($this->success) { ?>
2  <div class="alert alert-success">Produkt hinzugefügt!</div>
3  <?php } ?>
4
5  <?php if ($this->insertError) { ?>
6  <div class="alert alert-error">
7      Produkt konnte nicht hinzugefügt werden.
8  </div>
9  <?php } ?>
10
11 <?php
12 $this->form->prepare();
13 echo $this->form()->openTag($this->form);
14 echo $this->formRowTwb($this->form->get('product')->get('id'));
15 echo $this->formRowTwb($this->form->get('product')->get('name'));
16 echo $this->formRowTwb($this->form->get('product')->get('stock'));
17 echo $this->formSubmitTwb($this->form->get('submit'));
18 echo $this->form()->closeTag();
```

Listing 26.27

Nun noch kurz einen weiteren Test hinzufügen, um die Situation abzubilden, in der die Entity nicht persistiert werden kann. In diesem Zuge räume ich auch den AdminControllerTest ein wenig auf und lagere insb. die Erzeugung der Fake-Objekte in Hilfsfunktionen aus:

```
1  <?php
2  namespace ZfDeals\ControllerTest;
3
4  use ZfDeals\Controller\AdminController;
5  use Zend\Http\Request;
6  use Zend\Http\Response;
7  use Zend\Mvc\MvcEvent;
8  use Zend\Mvc\Router\RouteMatch;
9
10 class AdminControllerTest extends \PHPUnit_Framework_TestCase
11 {
12     private $controller;
13     private $request;
14     private $response;
```

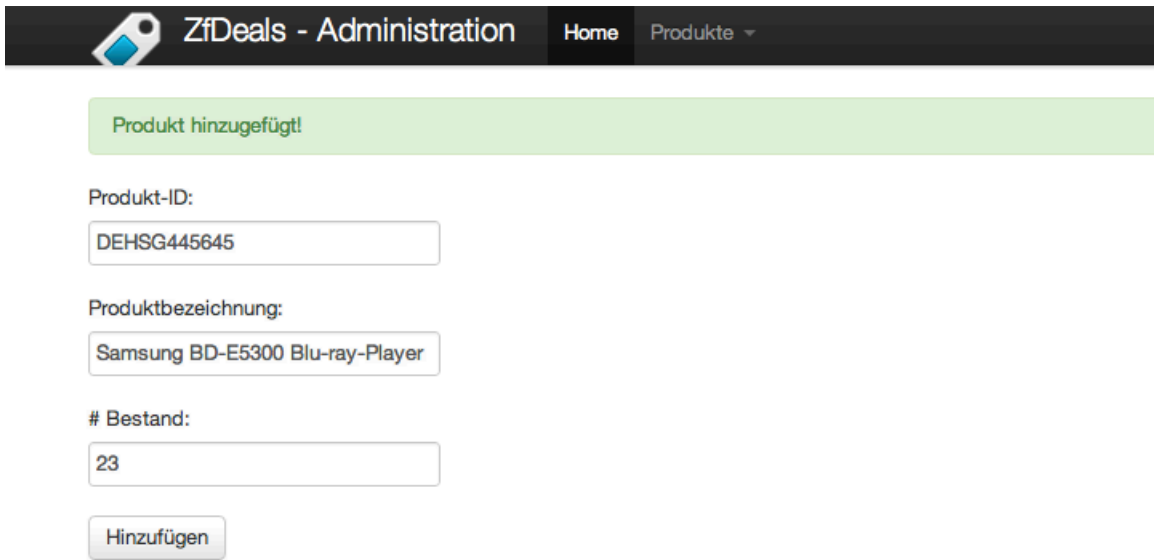
```
15     private $routeMatch;
16     private $event;
17
18     public function setUp()
19     {
20         $this->controller = new AdminController();
21         $this->request = new Request();
22         $this->response = new Response();
23         $this->routeMatch = new RouteMatch(array('controller' => 'admin'));
24         $this->routeMatch->setParam('action', 'add-product');
25         $this->event = new MvcEvent();
26         $this->event->setRouteMatch($this->routeMatch);
27         $this->controller->setEvent($this->event);
28     }
29
30     public function testShowFormOnGetRequest()
31     {
32         $fakeForm = new \Zend\Form\Form('fakeForm');
33         $this->controller->setProductAddForm($fakeForm);
34         $this->request->setMethod('get');
35         $response = $this->controller->dispatch($this->request);
36         $viewModelValues = $response->getVariables();
37         $formReturned = $viewModelValues['form'];
38         $this->assertEquals($formReturned->getName(), $fakeForm->getName());
39     }
40
41     public function testShowFormOnValidationError()
42     {
43         $fakeForm = $this->getFakeForm(false);
44         $this->controller->setProductAddForm($fakeForm);
45         $this->request->setMethod('post');
46         $response = $this->controller->dispatch($this->request);
47         $viewModelValues = $response->getVariables();
48         $formReturned = $viewModelValues['form'];
49         $this->assertEquals($formReturned->getName(), $fakeForm->getName());
50     }
51
52     public function testCallMapperOnFormValidationSuccessPersistenceSuccess()
53     {
54         $fakeForm = $this->getFakeForm();
55
56         $fakeForm->expects($this->once())
```

```
57         ->method('getData')
58         ->will($this->returnValue(new \stdClass()));
59
60     $fakeMapper = $this->getFakeMapper();
61
62     $fakeMapper->expects($this->once())
63         ->method('insert')
64         ->will($this->returnValue(true));
65
66     $this->controller->setProductAddForm($fakeForm);
67     $this->controller->setProductMapper($fakeMapper);
68     $this->request->setMethod('post');
69     $response = $this->controller->dispatch($this->request);
70     $viewModelValues = $response->getVariables();
71     $this->assertTrue(isset($viewModelValues['success']));
72 }
73
74 public function testCallMapperOnFormValidationSuccessPersistenceError()
75 {
76     $fakeForm = $this->getFakeForm();
77
78     $fakeForm->expects($this->once())
79         ->method('getData')
80         ->will($this->returnValue(new \stdClass()));
81
82     $fakeMapper = $this->getFakeMapper();
83
84     $fakeMapper->expects($this->once())
85         ->method('insert')
86         ->will($this->throwException(new \Exception));
87
88     $this->controller->setProductAddForm($fakeForm);
89     $this->controller->setProductMapper($fakeMapper);
90     $this->request->setMethod('post');
91     $response = $this->controller->dispatch($this->request);
92     $viewModelValues = $response->getVariables();
93     $this->assertTrue(isset($viewModelValues['form']));
94     $this->assertTrue(isset($viewModelValues['insertError']));
95 }
96
97 public function getFakeForm($isValid = true)
98 {
```

```
99         $fakeForm = $this->getMock(  
100             'Zend\Form\Form', array('isValid', 'getData')  
101         );  
102  
103         $fakeForm->expects($this->once())  
104             ->method('isValid')  
105             ->will($this->returnValue($isValid));  
106  
107         return $fakeForm;  
108     }  
109  
110     public function getFakeMapper()  
111     {  
112         $fakeMapper = $this->getMock('ZfDeals\Mapper\Product',  
113             array('insert'),  
114             array(),  
115             '',  
116             false  
117         );  
118  
119         return $fakeMapper;  
120     }  
121 }
```

Listing 26.28

Und so sieht das Ganze im Moment dann aus:



The screenshot shows the 'ZfDeals - Administration' interface. At the top, there is a dark navigation bar with a logo and links for 'Home' and 'Produkte'. Below this, a green message box states 'Produkt hinzugefügt!'. The main form contains three input fields: 'Produkt-ID:' with the value 'DEHSG445645', 'Produktbezeichnung:' with the value 'Samsung BD-E5300 Blu-ray-Player', and '# Bestand:' with the value '23'. A 'Hinzufügen' button is located at the bottom of the form.

ZfDeals - Produkt hinzufügen nach dem erfolgreichen Absenden

Damit ist die Arbeit für diesen Sprint getan!

26.5 Sprint 3 - Deal anlegen, Deals anzeigen



Quellcode-Download

Den Quellcode nach Abschluss dieses Sprints findest du zum Download im Tag "Sprint3" bei [github](https://github.com/michael-romer/ZfDealsApp/tree/Sprint3)¹³.

Die erste User Story dieses Sprints lautet:

"Damit ich ein Produkt als Deal anbieten kann, möchte ich als Händler, dass ich einen Deal im System anlegen kann."

Das Akzeptanzkriterium dazu lautet:

- Über ein Formular lässt sich zu einem bestehenden Produkt ein Deal-Preis festlegen, der von einem definierbaren Startdatum bis zu einem definierbaren Enddatum für das jeweilige Produkt gilt.

¹³<https://github.com/michael-romer/ZfDealsApp/tree/Sprint3>

Die nächste User Story in diesem Sprint lautet:

“Damit ich ein Produkt kaufen kann, möchte ich als Käufer, dass auf einer Übersichtsseite alle aktuell verfügbaren Deals angezeigt werden.”

Die Akzeptanzkriterien lauten:

- Es werden alle Deals angezeigt, die aufgrund der zeitlichen Konfiguration gerade “aktiv” sind.
- Es werden nur die aktiven Deals angezeigt, bei denen es für das jeweils angebotene Produkt einen Warenbestand > 0 gibt.

Bevor ich mich diesen fachlichen Aufgaben widme, kümmere ich mich noch um einige infrastrukturelle Dinge.

Coding-Standard

Das Zend Framework 2 hält sich mit seinem Source Code an den [PSR-2 Coding-Standard](#)¹⁴. Ich will versuchen, ihn auch bei ZfDeals einzuhalten, so kann ich sicher gehen, dass sich später andere Entwickler im Quellcode zuhause fühlen, wenn sie ihn näher verstehen, anpassen oder weiterentwickeln müssen. Um sicherzustellen, dass ich mich immer an den Standard halte, Sorge ich dafür, dass auf meinem System das Tool [PHP_Codesniffer](#)¹⁵ in der aktuellen Version zur Verfügung steht. Auf meiner virtuellen Maschine ist “PHP_CodeSniffer” bereits verfügbar, bei Bedarf kann man das Tool wie folgt via [PEAR](#)¹⁶ installieren:

```
1 $ pear install PHP_CodeSniffer-1.3.6
```

Wenn ich nun das Kommando

```
1 $ phpcs -v --standard=psr2 ZfDeals/
```

im Verzeichnis `modules` ausführe, testet “PHP_CodeSniffer” automatisch den Code des ZfDeals-Moduls auf Konformität mit dem “PSR-2 Coding-Standard”. Auf Basis des aktuellen Codes stellt “PHP_CodeSniffer” eine ganze Menge Abweichungen vom Standard fest, z.B. in der `Module.php`:

¹⁴<https://github.com/pmjones/fig-standards/blob/psr-1-style-guide/proposed/PSR-2-advanced.md>

¹⁵http://pear.php.net/package/PHP_CodeSniffer/download/

¹⁶<http://pear.php.net/>

```
1 FILE: /vagrant/module/ZfDeals/Module.php
2 -----
3 FOUND 7 ERROR(S) AFFECTING 3 LINE(S)
4 -----
5 11 | ERROR | Opening parenthesis of a multi-line
6 function call must be the last content on the line
7
8 11 | ERROR | Only one argument is allowed per line
9 in a multi-line function call
10
11 11 | ERROR | Only one argument is allowed per line
12 in a multi-line function call
13
14 11 | ERROR | Expected 1 space after FUNCTION keyword; 0 found
15
16 14 | ERROR | Only one argument is allowed per line
17 in a multi-line function call
18
19 14 | ERROR | Closing parenthesis of a multi-line function
20 call must be on a line by itself
21
22 32 | ERROR | Expected 1 blank line at end of file; 0 found
```

Nachdem ich alle Probleme beseitigt habe, gibt “PHP_CodeSniffer” positives Feedback durch verbale Zurückhaltung. In meiner [münsterländischen Heimat](http://www.muensterland.de/)¹⁷ gibt es so eine Redensart, die auch die Menschen hier ganz gut beschreibt: “Etwas ist gut, wenn keiner meckert”. So ungefähr verhält es sich also auch beim “PHP_CodeSniffer”. Wenn nach Ausführen des Kommandos kaum etwas auf der Shell zu sehen ist, dann ist das ein gutes Zeichen.



Coding-Standard-Probleme automatisch beheben lassen

Mit “PHP-CS-Fixer” stellt Fabien Potencier übrigens ein hilfreiches Tool bereit, mit dem sich viele der üblichen Coding-Standard-Probleme ohne manuelles Zutun beheben lassen.

Dass ich nach den Umbauten jetzt einfach meine Testbatterie laufen lassen kann und feststelle, dass ich bei den Anpassungen nicht versehentlich etwas kaputt gemacht habe, gibt mir ein gutes Gefühl.

¹⁷<http://www.muensterland.de/>

Damit ich demnächst so einfach wie möglich und kontinuierlich den Quellcode auf Standardkonformität testen kann, erstelle ich im Verzeichnis `tests` das kleine Script `checkstyle.php`:

```
1 <?php
2 echo shell_exec('phpcs --standard=psr2 ../module/ZfDeals') . PHP_EOL;
```

Listing 26.29

Demnächst kann ich die Überprüfung mit dem folgenden, etwas leichter zu merkenden Kommando ausführen:

```
1 $ php checkstyle.php
```

Datenbank-Schema hinterlegen

Damit man schnell und unkompliziert die für “ZfDeals” notwendige Tabellenstruktur erzeugen kann, lege ich unter `/module/ZfDeals/data` noch das passende DDL-File `structure.sql` an, auf dessen Basis leicht die notwendigen Tabellen erzeugt werden können:

```
1 CREATE TABLE product(
2     id varchar(255) NOT NULL,
3     name varchar(255) NOT NULL,
4     stock int(10) NOT NULL, PRIMARY KEY (id)
5 );
```

Weitere Tabellen werden sicherlich bereits in Kürze hinzukommen.

Deal-Entity

Jetzt aber zu den User Stories des Sprints. Ich erstelle zunächst einmal die `Deal-Entity`

```
1 <?php
2 namespace ZfDeals\Entity;
3
4 class Deal
5 {
6     protected $id;
7     protected $price;
8     protected $startDate;
9     protected $endDate;
10    protected $product;
```



```
11
12     public function setEndDate($endDate)
13     {
14         $this->endDate = $endDate;
15     }
16
17     public function getEndDate()
18     {
19         return $this->endDate;
20     }
21
22     public function setStartDate($startDate)
23     {
24         $this->startDate = $startDate;
25     }
26
27     public function getStartDate()
28     {
29         return $this->startDate;
30     }
31
32     public function setId($id)
33     {
34         $this->id = $id;
35     }
36
37     public function getId()
38     {
39         return $this->id;
40     }
41
42     public function setPrice($price)
43     {
44         $this->price = $price;
45     }
46
47     public function getPrice()
48     {
49         return $this->price;
50     }
51
52     public function setProduct($product)
```

```

53     {
54         $this->product = $product;
55     }
56
57     public function getProduct()
58     {
59         return $this->product;
60     }
61 }

```

Listing 26.30

Ein Deal ist demnach charakterisiert durch einen (Sonder-)preis, ein Startdatum, Enddatum und eine Referenz zum jeweiligen Produkt, auf das sich der Deal bezieht. Die erforderliche Datenstruktur, die ich auch in die `structure.sql` aufnehme, sieht wie folgt aus:

```

1 CREATE TABLE deal(
2     id int(10) NOT NULL AUTO_INCREMENT,
3     price float NOT NULL,
4     startDate date NOT NULL,
5     endDate date NOT NULL,
6     product varchar(255) NOT NULL,
7     PRIMARY KEY (id)
8 );

```

Einen Deal erfassen

Ich erweitere das Menü der Administration um die Sektion “Deals” mit dem Menüpunkt “Deal hinzufügen”, der auf die URL `/deals/admin/deal/add` führt. Wie gehabt lege ich dazu zunächst die notwendige Route in der `module.config.php` an. Das Formular `DealAdd` kommt von nun an für die Erfassung von Deals zum Einsatz:

```

1 <?php
2 namespace ZfDeals\Form;
3
4 use Zend\Form\Form;
5 use Zend\ServiceManager\ServiceManager;
6 use Zend\ServiceManager\ServiceManagerAwareInterface;
7
8 class DealAdd extends Form
9 {
10     public function __construct()

```

```

11     {
12         parent::__construct('dealAdd');
13         $this->setAttribute('action', '/deals/admin/deal/add');
14         $this->setAttribute('method', 'post');
15
16         $this->add(
17             array(
18                 'type' => 'ZfDeals\Form\DealFieldset',
19                 'options' => array(
20                     'use_as_base_fieldset' => true
21                 )
22             )
23         );
24
25         $this->add(
26             array(
27                 'name' => 'submit',
28                 'attributes' => array(
29                     'type' => 'submit',
30                     'value' => 'Hinzufügen'
31                 ),
32             )
33         );
34     }
35 }

```

Listing 26.31

Und hier das DealFieldset:

```

1  <?php
2  namespace ZfDeals\Form;
3
4  use Zend\Form\Fieldset;
5  use Zend\InputFilter\InputFilterInterface;
6
7  class DealFieldset extends Fieldset
8  {
9      public function __construct()
10     {
11         parent::__construct('deal');
12
13         $this->add(

```

```
14         array(
15             'name' => 'product',
16             'type' => 'ZfDeals\Form\ProductSelectorFieldset',
17         )
18     );
19
20     $this->add(
21         array(
22             'name' => 'price',
23             'type' => 'Zend\Form\Element\Number',
24             'attributes' => array(
25                 'step' => 'any'
26             ),
27             'options' => array(
28                 'label' => 'Preis:',
29             )
30         )
31     );
32
33     $this->add(
34         array(
35             'name' => 'startDate',
36             'type' => 'Zend\Form\Element\Date',
37             'options' => array(
38                 'label' => 'Startdatum:'
39             ),
40         )
41     );
42
43     $this->add(
44         array(
45             'name' => 'endDate',
46             'type' => 'Zend\Form\Element\Date',
47             'options' => array(
48                 'label' => 'Enddatum:'
49             ),
50         )
51     );
52 }
53 }
```

Listing 26.32

Das DealFieldset enthält selbst noch das ProductSelectorFieldset, das zur Auswahl des betreffenden Produktes dient:

```
1  <?php
2  namespace ZfDeals\Form;
3
4  use Zend\Form\Fieldset;
5  use Zend\InputFilter\InputFilterInterface;
6
7  class ProductSelectorFieldset extends Fieldset
8  {
9      public function __construct()
10     {
11         parent::__construct('productSelector');
12         $this->setHydrator(new\Zend\Stdlib\Hydrator\Reflection());
13         $this->setObject(new \ZfDeals\Entity\Product());
14
15         $this->add(
16             array(
17                 'name' => 'id',
18                 'type'  => 'Zend\Form\Element\Select',
19                 'options' => array(
20                     'label' => 'Produkt-ID:',
21                     'value_options' => array(
22                         '1' => 'Label 1',
23                         '2' => 'Label 2',
24                     ),
25                 ),
26             )
27         );
28     }
29 }
```

Listing 26.33

Das entsprechende Formularfeld ist zunächst mit Dummy-Werten gefüllt, die später dann im Controller dynamisch durch die “echten” Werte ersetzt werden.



INTL-Extension benötigt

Beim ersten Aufruf des Formulars bekomme ich einen “PHP Fatal error: Class ‘NumberFormatter’ not found”. Der Grund dafür ist das Fehlen der INTL-Extension, die das Zend Framework 2 zwingend erfordert - auch dann, wenn man nicht direkt mit den I18N-Funktionen des Frameworks arbeitet. Auf einem Linux-System lässt sie sich etwa einfach mit dem Kommando `apt-get install php5-intl` installieren.

Die für die Verarbeitung des Formulars notwendige Action bringe ich im AdminController des Moduls unter. Die AdminControllerFactory erweitere ich entsprechend, so dass auch das DealAdd-Formular injiziert wird:

```

1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class AdminControllerFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {
11         $ctr = new AdminController();
12         $productAddForm = new \ZfDeals\Form\ProductAdd();
13         $productAddForm->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
14         $productAddForm->bind(new \ZfDeals\Entity\Product());
15         $ctr->setProductAddForm($productAddForm);
16
17         $mapper = $serviceLocator->getServiceLocator()
18             ->get('ZfDeals\Mapper\Product');
19
20         $ctr->setProductMapper($mapper);
21         $dealAddForm = new \ZfDeals\Form\DealAdd();
22         $ctr->setDealAddForm($dealAddForm);
23
24         $dealAddForm
25             ->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
26
27         $dealAddForm->bind(new \ZfDeals\Entity\Deal());
28     }

```

```

29         $dealMapper = $serviceLocator->getServiceLocator()
30             ->get('ZfDeals\Mapper\Deal');
31
32         $ctr->setDealMapper($dealMapper);
33         return $ctr;
34     }
35 }

```

Listing 26.34

In der `addDealAction` im `AdminController` wird dann das Formular verarbeitet. Dazu fülle ich zu Anfang den `ProductSelectorFieldset` mit den Daten der tatsächlich im System vorhandenen Produkte. Über den `DealMapper` werden neue Deals in die Datenbank geschrieben und später auch die aktiven Deals ermittelt:

```

1  <?php
2
3  namespace ZfDeals\Mapper;
4
5  use ZfDeals\Entity\Deal as DealEntity;
6  use Zend\Stdlib\Hydrator\HydratorInterface;
7  use Zend\Db\TableGateway\TableGateway;
8  use Zend\Db\TableGateway\Feature\RowGatewayFeature;
9  use Zend\Db\Sql\Sql;
10 use Zend\Db\Sql\Insert;
11
12 class Deal extends TableGateway
13 {
14     protected $tableName = 'deal';
15     protected $idCol = 'id';
16     protected $entityPrototype = null;
17     protected $hydrator = null;
18
19     public function __construct($adapter)
20     {
21         parent::__construct(
22             $this->tableName,
23             $adapter,
24             new RowGatewayFeature($this->idCol)
25         );
26
27         $this->entityPrototype = new DealEntity();
28         $this->hydrator = new \Zend\Stdlib\Hydrator\Reflection;

```

```
29     }
30
31     public function insert($entity)
32     {
33         return parent::insert($this->hydrator->extract($entity));
34     }
35
36     public function findActiveDeals()
37     {
38         $sql = new \Zend\Db\Sql\Sql($this->getAdapter());
39         $select = $sql->select()
40             ->from($this->tableName)
41             ->join('product', 'deal.product=product.id')
42             ->where('DATE(startDate) <= DATE(NOW())')
43             ->where('DATE(endDate) >= DATE(NOW())')
44             ->where('stock > 0');
45
46         $stmt = $sql->prepareStatementForSqlObject($select);
47         $results = $stmt->execute();
48
49         return $this->hydrate($results);
50     }
51
52     public function hydrate($results)
53     {
54         $deals = new \Zend\Db\ResultSet\HydratingResultSet(
55             $this->hydrator,
56             $this->entityPrototype
57         );
58
59         return $deals->initialize($results);
60     }
61 }
```

Listing 26.35

Die addDealAction ist grundsätzlich zwar noch recht schlank, aber bereits jetzt nicht mehr so wirklich gut überschaubar:


```
1  <?php
2  // [...]
3  public function addDealAction()
4  {
5      $form = $this->dealAddForm;
6
7      $products = $this->productMapper->select();
8      $fieldElements = array();
9
10     foreach ($products as $product) {
11         $fieldElements[$product['id']] = $product['name'];
12     }
13
14     $form->get('deal')->get('product')
15         ->get('id')->setValueOptions($fieldElements);
16
17     if ($this->getRequest()->isPost()) {
18         $form->setData($this->getRequest()->getPost());
19
20         if ($form->isValid()) {
21             $model = new ViewModel(
22                 array(
23                     'form' => $form
24                 )
25             );
26
27             $newDeal = $form->getData();
28             $newDeal->setProduct($newDeal->getProduct()->getId());
29
30             try {
31                 $this->dealMapper->insert($newDeal);
32                 $model->setVariable('success', true);
33             } catch (\Exception $e) {
34                 $model->setVariable('insertError', true);
35             }
36
37             return $model;
38         } else {
39             return new ViewModel(
40                 array(
41                     'form' => $form
42                 )
43             );
44         }
45     }
46 }
```

```

43         );
44     }
45     } else {
46         return new ViewModel(
47             array(
48                 'form' => $form
49             )
50         );
51     }
52 }
53 // [...]

```

Listing 26.36

Aktive Deals anzeigen

Für die Anzeige der aktiven Deals erstelle ich den IndexController, in der passenden IndexControllerFactory injiziere ich die Mapper Deal und Product, um die für die Anzeige notwendigen Daten aus der Datenbank zu laden:

```

1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6  use Zend\Form\Annotation\AnnotationBuilder;
7
8  class IndexController extends AbstractActionController
9  {
10     private $dealMapper;
11     private $productMapper;
12
13     public function indexAction()
14     {
15         $deals = $this->dealMapper->findActiveDeals();
16         $dealsView = array();
17
18         foreach ($deals as $deal) {
19             $deal->setProduct(
20                 $this->productMapper->findOneById($deal->getProduct())
21             );
22

```

```
23         $dealsView[] = $deal;
24     }
25
26     return new ViewModel(
27         array(
28             'deals' => $dealsView
29         )
30     );
31 }
32
33 public function setDealMapper($dealMapper)
34 {
35     $this->dealMapper = $dealMapper;
36 }
37
38 public function getDealMapper()
39 {
40     return $this->dealMapper;
41 }
42
43 public function setProductMapper($productMapper)
44 {
45     $this->productMapper = $productMapper;
46 }
47
48 public function getProductMapper()
49 {
50     return $this->productMapper;
51 }
52 }
```

Listing 26.37

Die `foreach`-Schleife ist in diesem Fall zu verschmerzen, weil die Datenmenge zunächst wohl überschaubar ist, stellt aber grundsätzlich eine “Performancebremse” dar. Ich werde das später noch überarbeiten müssen.

Für die Darstellung der Deals verwende ich ein eigenes Layout, dass ich in der `Module`-Klasse in der `init`-Methode aktiviere:

```
1  <?php
2  public function init(ModuleManager $moduleManager)
3  {
4      $sharedEvents = $moduleManager->getEventManager()->getSharedManager();
5
6      $sharedEvents->attach(
7          'ZfDeals\Controller\AdminController',
8          'dispatch',
9          function ($e) {
10             $controller = $e->getTarget();
11             $controller->layout('zf-deals/layout/admin');
12         },
13         100
14     );
15
16     $sharedEvents->attach(
17         'ZfDeals\Controller\IndexController',
18         'dispatch',
19         function ($e) {
20             $controller = $e->getTarget();
21             $controller->layout('zf-deals/layout/site');
22         },
23         100
24     );
25 }
```

Listing 26.38

Unter der URL /deals werden nun alle aktiven, zuvor über die Administration hinzugefügten Deals angezeigt.

Eine eigene Art von Controllern für die Formularverarbeitung

So richtig glücklich bin ich allerdings nicht. Da wäre zunächst einmal der AdminController. Dort habe ich die beiden “add-Actions”, einmal die für das Product und einmal die für den Deal. Beide Methoden sind sich inhaltlich sehr ähnlich, weil sie beide ein Formular verarbeiten. Auch sind sie schon jetzt recht unübersichtlich, müssen sie sich doch sowohl um die initiale Darstellung des Formulars, dessen Validierung samt erneuter Anzeige bei fehlerhafter Eingabe, sowie die eigentliche Verarbeitung valider Daten kümmern. Ich habe mich bereits dabei erwischt, wie ich den “Verarbeitungsrumpf” der addDealAction von der addProductAction via Copy&Paste übernommen und modifiziert habe - ein klares Zeichen dafür, dass hier Optimierungsbedarf besteht. Ein weiteres Problem ist die Tatsache, dass ich im Rahmen der IndexControllerFactory immer beide Formulare

injiziere, obwohl ja immer nur eines der beiden benötigt wird. Das erscheint mir doch etwas unlogisch. Zudem gibt es im passenden View-File zu viel Darstellungslogik.

Um diese Probleme auf einen Schlag zu beheben, mache ich mir die Tatsache zunutze, dass ein Controller im Zend Framework 2 grundsätzlich lediglich über eine `onDispatch()`-Methode verfügen muss, die aus einem Request eine Response macht. Hält man sich daran, kann man einen Typ von Controller realisieren, der sich nicht an das standardmäßige "Controller/Action-Schema" hält, sondern die Dinge vollkommen anders angeht. Und genau das mache ich hier. Ich modelliere einen Controller, der speziell für die Verarbeitung eines Formulars gemacht ist, den `AbstractFormController`:

```
1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\Mvc\Controller\AbstractController;
5  use Zend\Mvc\MvcEvent;
6  use Zend\Form\Form as Form;
7  use Zend\View\Model\ViewModel;
8
9  abstract class AbstractFormController extends AbstractController
10 {
11     protected $form;
12
13     public function __construct(Form $form)
14     {
15         $this->form = $form;
16     }
17
18     public function onDispatch(MvcEvent $e)
19     {
20         if (method_exists($this, 'prepare')) {
21             $this->prepare();
22         }
23
24         $routeMatch = $e->getRouteMatch();
25
26         if ($this->getRequest()->isPost()) {
27             $this->form->setData($this->getRequest()->getPost());
28
29             if ($this->form->isValid()) {
30                 $routeMatch->setParam('action', 'process');
31                 $return = $this->process();
32             } else {
```

```
33             $routeMatch->setParam('action', 'error');
34             $return = $this->error();
35         }
36
37     } else {
38         $routeMatch->setParam('action', 'show');
39         $return = $this->show();
40     }
41
42     $e->setResult($return);
43     return $return;
44 }
45
46 abstract protected function process();
47
48 protected function show()
49 {
50     return new ViewModel(
51         array(
52             'form' => $this->form
53         )
54     );
55 }
56
57 protected function error()
58 {
59     return new ViewModel(
60         array(
61             'form' => $this->form
62         )
63     );
64 }
65
66 public function setForm($form)
67 {
68     $this->form = $form;
69 }
70
71 public function getForm()
72 {
73     return $this->form;
74 }
```

75 }

Listing 26.39

Er funktioniert wie folgt: Wenn im Rahmen des Routings ein Controller ermittelt wurde, der auf dem `AbstractFormController` basiert, so wird wie gehabt dessen `dispatch()`-Methode und dann ereignisgesteuert dessen `onDispatch()`-Methode durch das Framework aufgerufen. Dort wird aber nicht, wie sonst üblich, der im Rahmen des Routings ermittelte Wert für eine etwaige Action ausgelesen, sondern stattdessen auf Basis des Status der Formulars entschieden, an welcher Stelle die Verarbeitung fortgesetzt wird. Wird die URL über einen GET-Request aufgerufen, so ist klar, dass wir an jener Stelle der Formularinteraktion sind, an der das Formular initial angezeigt werden soll. Daher wird die Methode `show()` aufgerufen, die im `AbstractFormController` sogar bereits rudimentär implementiert ist. Ein konkreter Controller, der von `AbstractFormController` ableitet, könnte zur Darstellung also vollkommen ohne eine eigene `show()`-Methode auskommen, falls es keine speziellen Anforderungen gibt. Der `AbstractFormController` ist so konzipiert, dass bei Instanziierung des jeweils auf dem `AbstractFormController` basierenden, konkreten Controllers, das Formular, um das es sich dreht, übergeben werden muss. So kann im Rahmen der Methode `dispatch()` und bei einem POST-Request die "isValid-Überprüfung" automatisch angestoßen werden, weil das betreffende Formular auf jeden Fall vorliegt. Je nach Ergebnis wird dann die Methode `process()` oder `error()` aufgerufen. Weil die eigentliche Verarbeitung valider Formulare wohl immer recht spezifisch für das eigentliche Formular ist, bringt der `AbstractFormController` nur eine abstrakte Methode `process()` mit, die entsprechend implementiert werden muss. Kurz gesagt: Der `AbstractFormController` befreit uns vom "Spaghetticode"¹⁸ für die Formularbearbeitung, der andernfalls notwendig und meist in eine Action gepresst ist.

Noch eine Anmerkung: Die `setParam()`-Aufrufe, die schlussendlich dann doch einen Wert für Action setzen, der für den `AbstractFormController` ja nicht mehr im Routing gesetzt wird, bzw. gesetzt werden kann, ist lediglich dafür gedacht, dass zu einem späteren Zeitpunkt für das Rendering ein passendes View-File eingelesen werden kann. Das bedeutet, dass die View-Files `show.phtml`, `process.phtml` und `error.phtml` in View-Verzeichnis des Controllers vorliegen müssen, damit der `AbstractFormController` ordnungsgemäß funktioniert.

Die Verarbeitung der beiden Formulare lagere ich nun jeweils in eigene Controller aus, die von `AbstractFormController` ableiten. Der `ProductAddFormController` stellt sich wie folgt dar:

¹⁸<http://de.wikipedia.org/wiki/Spaghetticode>

```
1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6  use Zend\Stdlib\Hydrator\Reflection;
7  use ZfDeals\Entity\Product as ProductEntity;
8  use ZfDeals\Form\ProductAdd as ProductAddForm;
9
10 class ProductAddFormController extends AbstractFormController
11 {
12     private $productMapper;
13
14     public function __construct(ProductAddForm $form)
15     {
16         parent::__construct($form);
17     }
18
19     public function prepare()
20     {
21         $this->form->setHydrator(new Reflection());
22         $this->form->bind(new ProductEntity());
23     }
24
25     public function process()
26     {
27         $model = new ViewModel(
28             array(
29                 'form' => $this->form
30             )
31         );
32
33         try {
34             $this->productMapper->insert($this->form->getData());
35             $model->setVariable('success', true);
36         } catch (\Exception $e) {
37             $model->setVariable('insertError', true);
38         }
39
40         return $model;
41     }
42 }
```



```
43     public function setProductMapper($productMapper)
44     {
45         $this->productMapper = $productMapper;
46     }
47
48     public function getProductMapper()
49     {
50         return $this->productMapper;
51     }
52 }
```

Listing 26.40

Der DealAddFormController wiederum so:

```
1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\Mvc\Controller\AbstractActionController;
5  use Zend\View\Model\ViewModel;
6  use Zend\Stdlib\Hydrator\Reflection;
7  use ZfDeals\Entity\Deal as DealEntity;
8
9  class DealAddFormController extends AbstractFormController
10 {
11     private $productMapper;
12     private $dealMapper;
13
14     public function prepare()
15     {
16         $this->form->setHydrator(new Reflection());
17         $this->form->bind(new DealEntity());
18
19         $products = $this->productMapper->select();
20         $fieldElements = array();
21
22         foreach ($products as $product) {
23             $fieldElements[$product['id']] = $product['name'];
24         }
25
26         $this->form->get('deal')
27             ->get('product')
28             ->get('id')->setValueOptions($fieldElements);
```

```
29     }
30
31     public function process()
32     {
33         $model = new ViewModel(
34             array(
35                 'form' => $this->form
36             )
37         );
38
39         $newDeal = $this->form->getData();
40         $newDeal->setProduct($newDeal->getProduct()->getId());
41
42         try {
43             $this->dealMapper->insert($newDeal);
44             $model->setVariable('success', true);
45         } catch (\Exception $e) {
46             $model->setVariable('insertError', true);
47         }
48
49         return $model;
50     }
51
52     public function setProductMapper($productMapper)
53     {
54         $this->productMapper = $productMapper;
55     }
56
57     public function getProductMapper()
58     {
59         return $this->productMapper;
60     }
61
62     public function setDealMapper($dealMapper)
63     {
64         $this->dealMapper = $dealMapper;
65     }
66
67     public function getDealMapper()
68     {
69         return $this->dealMapper;
70     }
```

```
71 }
```

Listing 26.41

Die Methode `prepare()` wird übrigens, insofern sie denn existiert, vom `AbstractFormController` standardmäßig aufgerufen, bevor mit der eigentlichen Formulareauswertung und -verarbeitung begonnen wird.

Unit-Tests für den `AbstractFormController`

Nun benötige ich auch nur noch für den `AbstractFormController`. Dort kann ich stellvertretend für alle formularverarbeitenden Controller die grundsätzliche Logik einmalig testen. Ich spare mir also auch eine ganze Menge Arbeit bei den Tests von “Form-Controllern”, die dann nicht immer auch mit Fake-Objekten wie `RouteMatch`, `Request`, usw. ausgestattet werden müssen:

```
1  <?php
2  namespace ZfDeals\ControllerTest;
3
4  use ZfDeals\Controller\AbstractFormController;
5  use Zend\Http\Request;
6  use Zend\Http\Response;
7  use Zend\Mvc\MvcEvent;
8  use Zend\Mvc\Router\RouteMatch;
9  use ZfDeals\Form\ProductAdd as ProductAddForm;
10
11 class AbstractFormControllerTest extends \PHPUnit_Framework_TestCase
12 {
13     private $controller;
14     private $request;
15     private $response;
16     private $routeMatch;
17     private $event;
18
19     public function setUp()
20     {
21         $fakeController = $this->getMockForAbstractClass(
22             'ZfDeals\Controller\AbstractFormController',
23             array(),
24             '',
25             false
26         );
27
```

```
28         $this->controller = $fakeController;
29         $this->request = new Request();
30         $this->response = new Response();
31
32         $this->routeMatch = new RouteMatch(
33             array('controller' => 'abstract-form')
34         );
35
36         $this->event = new MvcEvent();
37         $this->event->setRouteMatch($this->routeMatch);
38         $this->controller->setEvent($this->event);
39     }
40
41     public function testShowOnGetRequest()
42     {
43         $this->form = new \Zend\Form\Form('fakeForm');
44         $this->controller->setForm($this->form);
45         $this->request->setMethod('get');
46         $response = $this->controller->dispatch($this->request);
47         $viewModelValues = $response->getVariables();
48         $formReturned = $viewModelValues['form'];
49
50         $this->assertEquals(
51             $formReturned->getName(), $this->form->getName()
52         );
53     }
54
55     public function testErrorOnValidationError()
56     {
57         $fakeForm = $this->getMock(
58             'Zend\Form\Form', array('isValid')
59         );
60
61         $fakeForm->expects($this->once())
62             ->method('isValid')
63             ->will($this->returnValue(false));
64
65         $this->controller->setForm($fakeForm);
66         $this->request->setMethod('post');
67         $response = $this->controller->dispatch($this->request);
68         $viewModelValues = $response->getVariables();
69         $formReturned = $viewModelValues['form'];
```

```

70         $this->assertEquals($formReturned, $fakeForm);
71     }
72
73     public function testProcessOnValidationSuccess()
74     {
75         $fakeForm = $this->getMock(
76             'Zend\Form\Form', array('isValid')
77         );
78
79         $fakeForm->expects($this->once())
80             ->method('isValid')
81             ->will($this->returnValue(true));
82
83         $this->controller->setForm($fakeForm);
84         $this->request->setMethod('post');
85
86         $this->controller->expects($this->once())
87             ->method('process')
88             ->will($this->returnValue(true));
89
90         $response = $this->controller->dispatch($this->request);
91     }
92 }

```

Listing 26.42

Einfache Factories als Closure realisieren

Zudem habe ich in der Anwendung eine ganze Reihe von Factories, die allesamt keine wirklich nennenswerten Aufgaben (mehr) haben. So instanziert etwa die `DealAddFormControllerFactory` lediglich einige Services bzw. beschafft sich sie über den `ServiceManager`:

```

1  <?php
2  namespace ZfDeals\Controller;
3
4  use Zend\ServiceManager\FactoryInterface;
5  use Zend\ServiceManager\ServiceLocatorInterface;
6
7  class DealAddFormControllerFactory implements FactoryInterface
8  {
9      public function createService(ServiceLocatorInterface $serviceLocator)
10     {

```

```
11         $form = new \ZfDeals\Form\DealAdd();
12         $ctr = new DealAddFormController($form);
13
14         $dealMapper = $serviceLocator
15             ->getServiceLocator()->get('ZfDeals\Mapper\Deal');
16
17         $ctr->setDealMapper($dealMapper);
18
19         $productMapper = $serviceLocator->getServiceLocator()
20             ->get('ZfDeals\Mapper\Product');
21
22         $ctr->setProductMapper($productMapper);
23         return $ctr;
24     }
25 }
```

Listing 26.43

Um den Code weiter zu vereinfachen, überführe ich die Factories in Closures und füge sie direkt der Konfiguration des Moduls hinzu:

```
1  <?php
2  // [...]
3  'controllers' => array(
4      'invokables' => array(
5          'ZfDeals\Controller\Admin'
6          => 'ZfDeals\Controller\AdminController',
7      ),
8      'factories' => array(
9          'ZfDeals\Controller\DealAddForm' => function ($serviceLocator) {
10              $form = new ZfDeals\Form\DealAdd();
11              $ctr = new ZfDeals\Controller\DealAddFormController($form);
12
13              $dealMapper = $serviceLocator
14                  ->getServiceLocator()->get('ZfDeals\Mapper\Deal');
15
16              $ctr->setDealMapper($dealMapper);
17
18              $productMapper = $serviceLocator->getServiceLocator()
19                  ->get('ZfDeals\Mapper\Product');
20
21              $ctr->setProductMapper($productMapper);
22              return $ctr;
23          }
24      )
25  )
```

```
23     },
24     'ZfDeals\Controller\ProductAddForm' => function ($serviceLocator) {
25         $form = new \ZfDeals\Form\ProductAdd();
26         $ctr = new ZfDeals\Controller\ProductAddFormController($form);
27
28         $productMapper = $serviceLocator->getServiceLocator()
29             ->get('ZfDeals\Mapper\Product');
30
31         $ctr->setProductMapper($productMapper);
32         return $ctr;
33     },
34     'ZfDeals\Controller\Index' => function ($serviceLocator) {
35         $ctr = new ZfDeals\Controller\IndexController();
36
37         $productMapper = $serviceLocator->getServiceLocator()
38             ->get('ZfDeals\Mapper\Product');
39
40         $dealMapper = $serviceLocator->getServiceLocator()
41             ->get('ZfDeals\Mapper\Deal');
42
43         $ctr->setDealMapper($dealMapper);
44         $ctr->setProductMapper($productMapper);
45         return $ctr;
46     }
47 ),
48 ),
```

Listing 26.44

Das war's für diesen Sprint!

26.6 Sprint 4 - Bestellformular



Quellcode-Download

Den Quellcode nach Abschluss dieses Sprints findest du zum Download im Tag "Sprint4" bei [github](https://github.com/michael-romer/ZfDealsApp/tree/Sprint4)¹⁹.

¹⁹<https://github.com/michael-romer/ZfDealsApp/tree/Sprint4>

Bevor ich mich den Aufgaben des neuen Sprints widme, nutze ich die sog. “Slack Time”, die freie Zeit zwischen dem vergangenen Sprint und dem Start des Folgenden, um mir einen Überblick über die aktuelle Testabdeckung zu machen. Dazu erweitere ich die `phpunit.xml` um die Konfiguration für das “Logging”:

```
1 <phpunit bootstrap="./bootstrap.php">
2     <testsuites>
3         <testsuite name="AllTests">
4             <directory>./ZfDealsTest/FormTest</directory>
5             <directory>./ZfDealsTest/ControllerTest</directory>
6         </testsuite>
7     </testsuites>
8     <logging>
9         <log type="coverage-html"
10            target="./reports/coverage"
11            charset="UTF-8"
12            yui="true"
13            highlight="false"
14            lowUpperBound="35"
15            highLowerBound="70" />
16     </logging>
17 </phpunit>
```

Wenn ich nun das Kommando

```
1 $ phpunit
```

ausführe, so werden nicht mehr nur die Tests ausgeführt, sondern auch umfassende “Code Coverage Reports” im HTML-Format erzeugt, die Aufschluss darüber geben, welche Codestellen durch Testfälle überprüft werden und wie es grundsätzlich um die Testabdeckung bestellt ist:

```
1 ZfDeals: 55.16%
2     config: 0%
3     src: 67.51%
4         Controller: 90.41%
5         Entity: 25%
6         Form: 79.31%
7         Mapper: 0%
8     Module.php: 0%
```


Die Werte sind sicherlich nicht ganz optimal. Eine Testabdeckung von 75% oder mehr ist schon erstrebenswert und in einigen Bereichen gibt es noch gar keine Tests. Ich muss hier also auf jeden Fall noch nachbessern.

Eine Sache noch, bevor ich mich der User Story des Sprints kümmerge: Ich will von vornherein dafür sorgen, dass die Anwendung auf Mehrsprachigkeit ausgelegt und zumindest zum Start bereits Deutsch und Englisch verfügbar ist. Dafür lösche ich zunächst die Sprachdateien der ZendSkeletonApplication im Modul Application, die ich dort ja nicht mehr benötige. Damit ich auch die händisch erstellten Fehlermeldungen in den Formularen über Bord werfen und mich voll und ganz auf die Standard-Form-Elemente des Frameworks mit dessen vordefinierten Validatoren stützen kann, verwende ich als Ausgangspunkt für meine eigenen Sprachdateien die Datei `Zend_Validate.php` aus dem Verzeichnis `vendor/zendframework/zendframework/resources/languages/de/`. Diese enthält nämlich bereits alle deutschen Übersetzungen für die Standard-Validatoren, bzw. dessen Fehlermeldungen. Damit die Übersetzungen in Form eines PHP-Arrays vom Framework berücksichtigt werden, muss ich nun noch die Konfiguration des Translator in der `module.config.php` des Application-Moduls anpassen:

```
1  <?php
2  // [...]
3  'translator' => array(
4      'locale' => 'de_DE',
5      'translation_file_patterns' => array(
6          array(
7              'type'      => 'PhpArray',
8              'base_dir' => __DIR__ . '/../language',
9              'pattern'  => '%s.php',
10         ),
11     )
12 )
13 // [...]
```

Listing 26.45

Damit der konfigurierte Translator auch standardmäßig für in Validatoren herangezogen wird, passe ich noch die Methode `onBootstrap()` der Module-Klasse des Application-Moduls wie folgt an:

```
1  <?php
2  // [...]
3  public function onBootstrap($e)
4  {
5      \Zend\Validator\AbstractValidator::setDefaultTranslator(
6          $e->getApplication()->getServiceManager()->get('translator')
7      );
8
9      $eventManager      = $e->getApplication()->getEventManager();
10     $moduleRouteListener = new ModuleRouteListener();
11     $moduleRouteListener->attach($eventManager);
12 }
13 // [...]
```

Listing 26.46

Nun pflüge ich einmal durch die Layouts, Views und Formulare des ZfDeals-Moduls und Sorge dafür, dass der Translator an den notwendigen Stellen zum Einsatz kommt und das nun überflüssige, individuelle Fehlermeldungen in den Formularen entfernt sind.

Die User Story dieses Sprints lautet:

“Damit ich ein Produkt kaufen kann, möchte ich als Nutzer, dass mir ein Bestellformular zur Verfügung steht.”

Die Akzeptanzkriterien lauten:

- Die Auflistung der aktiven Deals ist pro Deal erweitert um einen “Kaufen-Button”.
- Über den “Kaufen-Button” gelangt man auf eine neue Seite mit einem Bestellformular, auf dem Name und Anschrift abgefragt werden. Alle Angaben sind verpflichtend.
- Die Administration ist erweitert um eine Aufstellung aller eingegangenen Bestellungen.

Hier gibt es gar nicht so viel Neues für mich. Für das Bestellformular lege ich die notwendigen Routen, das Formular und den Controller an. Die Order-Entity hält zukünftig die Bestelldaten und eine Referenz auf den Deal, der in Anspruch genommen wurde. Der zugehörige Order-Mapper sorgt für die Speicherung neuer Bestellungen. Der Checkout-Service wird der erste “Business Service” des Moduls. Er kümmert sich darum, dass eine neue Bestellung im System vermerkt und gleichzeitig der Bestand des bestellten Produkts um ein Stück reduziert wird. Ich kapsle diese Geschäftslogik in einem eigenen “Business Service”, weil ich diese Logik nicht im CheckoutFormController vorhalten möchte, denn so kann ich sie später etwa auch über einen Webservice oder über andere Controller zugänglich machen, und weil ich sie keiner Entity des Systems eindeutig zuordnen kann; es wird sowohl eine Bestellung erzeugt als auch ein bestehendes Produkt, bzw. dessen Warenbestand, manipuliert. Eine Entity wäre hier also kein logisch sinnvoller Ort. Die wesentliche Methode des CheckoutService ist process():

```
1  <?php
2  public function process($ordering)
3  {
4      try {
5          $this->orderMapper->insert($ordering);
6          $deal = $this->dealMapper->findOneById($ordering->getDeal());
7          $product = $this->productMapper->findOneById($deal->getProduct());
8
9          $this->productMapper->update(
10             array('stock' => $product->getStock() - 1),
11             array('productId' => $product->getProductId())
12         );
13
14     } catch (\Exception $e) {
15         throw new \DomainException('Order could not be processed.');
```

Listing 26.47

Der Code hier ist sehr einfach gehalten und fängt zum aktuellen Zeitpunkt beileibe nicht alle möglichen Ausnahmesituationen ab, keine Frage. Für den Moment belassen wir es aber mal dabei.

26.7 Sprint 5 - Modul auslagern

Die User Story dieses Sprints lautet:

“Damit ZfDeals von mir als Modul genutzt werden kann, möchte ich als Entwickler einer anderen Software, dass sich ‘ZfDeals’ über Composer in meine Zend Framework 2 Anwendung integrieren lässt.”

Die Akzeptanzkriterien lauten daher konsequenterweise:

- “ZfDeals” ist über Composer in eine andere Anwendung integrierbar.

Ein zusätzliches Repository für das Modul anlegen

Zunächst einmal lege ich ein weiteres Git-Repository an, in das ich alle Daten des Moduls überführe. Aus dem ursprünglichen Repository entferne ich dann entsprechend diese Daten, um im nächsten Schritt das neu angelegte Modul-Repository als Submodul in das ursprüngliche Repository einzubinden:

```
1 $ git submodule add https://github.com/michael-romer/ZfDeals module/ZfDeals
```

Das Kommando muss man auf oberster Verzeichnisebene ausführen und es erfordert, dass das Modul-Verzeichnis `ZfDeals` ebenfalls zuvor entfernt wurde. Auf diese Art und Weise habe ich mir jetzt auch eine Entwicklungsumgebung für das `ZfDeals`-Modul geschaffen: Im Bezug auf die Codeverwaltung habe ich das Modul von der eigentliche Anwendung separiert, so dass andere Entwickler den Code später einfach verwenden können. Gleichzeitig habe ich aber einen Workspace zur Verfügung, der neben dem Modul auch über eine Zend Framework-Application verfügt, in der ich das Modul überhaupt erst ausführen und entwickeln kann. Je nach dem, an welcher Stelle im Verzeichnisbaum meines Workspaces ich mich befinde, kann ich nun entweder ein Commit in das ursprüngliche Repository machen, oder aber separat in das Modul-Repository. Dazu muss ich lediglich die entsprechenden Git-Kommandos im Modul-Verzeichnis ausführen. Auf diese Weise lassen sich also ganz prima ZF2-Module entwickeln.

Beim Überführen der Modul-Daten in das Modul-Repository berücksichtige ich auch die Dateien im Verzeichnis `public` der Anwendung, die zuvor erzeugte Sprachdatei, sowie die Tests. Die lagen bislang ja irgendwo in der Anwendung selbst; wir wollen sie zukünftig ja aber im Rahmen des Moduls zu distribuieren. In diesem Zuge entferne ich auch den in der `module.config.php` konfigurierten DB-Adapter und verschiebe ihn für mich für die weitere Entwicklung in die Config der Anwendung. Das bedeutet, dass das Modul zukünftig selbst keine Datenbankverbindung herstellt, sondern von der "Host-Anwendung" erwartet, dass sie diese Aufgabe übernimmt und den entsprechenden Service dem Modul bereitstellt. Gleiches gilt auch für den Translator-Service. Damit das Modul über Composer installiert werden kann, erstelle ich das notwendig `composer.json`-File:

```
1 {
2     "name": "zf2book/zf-deals",
3     "description": "This is the companion to the
4         book 'Webentwicklung mit Zend Framework 2'",
5     "type": "library",
6     "keywords": [
7         "zfdeals"
8     ],
9     "homepage": "http://zendframework2.de",
10    "authors": [
11        {
12            "name": "Michael Romer",
13            "email": "zf2buch@michael-romer.de",
14            "homepage": "http://zendframework2.de"
15        }
16    ],
17    "require": {
18        "php": ">=5.3.3",
19        "zendframework/zendframework": "2.*"
```

```

20     },
21     "autoload": {
22         "psr-0": {
23             "ZfDeals": "src/"
24         },
25         "classmap": [
26             "./Module.php"
27         ]
28     }
29 }

```

Der Abschnitt `autoload` sorgt dafür, dass sowohl die Modul-Datei `Module.php`, als auch die Klassen unterhalb von `src` bei Bedarf automatisch geladen werden.

Bleibt nur noch, dass Modul als installierbares Paket bei [Packagist](https://packagist.org/packages/zf2book/zf-deals)²⁰ unter dem Namen `zf2book/zf-deals` zu registrieren, so dass es via Composer bezogen werden kann. Das Ganze ist eine Sache von wenigen Minuten. Die Datei `README.md` enthält die für den Anwender des Moduls notwendigen Installationsanweisungen:

```

1  Install
2  =====
3
4  Main Install
5  -----
6
7  1. Add the following statement to the requirements-block
8  of your composer.json: "zf2book/zf-deals": "dev-master",
9  "dlu/dlutwbootstrap": "dev-master"
10
11 2. Run a composer update to download the libraries needed.
12
13 3. Add "ZfDeals" and "DluTwBootstrap" to the list of active
14 modules in `application.config.php`
15
16 4. Import the SQL schema located in
17 `/vendor/zf2book/zf-deals/data/structure.sql`
18
19 5. Copy `/vendor/zf2book/zf-deals/data/public/zf-deals`
20 to the public folder of your application.
21
22 Post Install

```

²⁰<https://packagist.org/packages/zf2book/zf-deals>

```

23 -----
24
25 1. If you do not already have a valid
26 Zend\Db\Adapter\Adapter in your service manager configuration,
27 put the following in `/config/autoload/db.local.php`:
28
29     <?php
30
31     $dbParams = array(
32         'database' => 'changeme',
33         'username' => 'changeme',
34         'password' => 'changeme',
35         'hostname' => 'changeme',
36     );
37
38     return array(
39         'service_manager' => array(
40             'factories' => array(
41                 'Zend\Db\Adapter\Adapter' => function
42                     ($sm) use ($dbParams) {
43                     return new Zend\Db\Adapter\Adapter(array(
44                         'driver'      => 'pdo',
45                         'dsn'         =>
46 'mysql:dbname=' . $dbParams['database'] . ';host=' . $dbParams['hostname'],
47                         'database'   => $dbParams['database'],
48                         'username'   => $dbParams['username'],
49                         'password'   => $dbParams['password'],
50                         'hostname'   => $dbParams['hostname'],
51                     ));
52                 },
53             ),
54         ),
55     );
56
57 2. Navigate to http://yourproject/deals or http://yourproject/deals/admin

```

Wie man sieht, sind neben dem Bezug des Codes via Composer noch einige weitere Dinge zu tun, damit das Modul seine Arbeit aufnehmen kann:

- Sowohl ZfDeals, als auch DluTwBootstrap (dieses Modul nutzen wir ja wiederum in unserem Modul) müssen in der `application.config.php` aktiviert werden.
- Die notwendige Datenstruktur muss auf Basis der `structure.sql` in der Datenbank erzeugt werden.

- Die Bilder, CSS- und JS-Files, die unser Modul benötigt, müssen über den `public`-Folder der Anwendung bereitgestellt werden, in dem das entsprechende Verzeichnis aus dem Modul kopiert wird.

Die Modul-Konfiguration überschaubar halten

Die `module.config.php` von `ZfDeals` ist mittlerweile sehr unübersichtlich geworden, enthält sie doch neben den Routen-Definitionen u.a. auch die Konfiguration von Services und Controllern. Um die Lesbarkeit zu erhöhen, separiere ich die einzelnen Abschnitt voneinander. Die `services.config.php` enthält nur noch die Services des Moduls:

```
1  <?php
2  return array(
3      'factories' => array(
4          'ZfDeals\Mapper\Product' => function ($sm) {
5              return new \ZfDeals\Mapper\Product(
6                  $sm->get('Zend\Db\Adapter\Adapter')
7              );
8          },
9          'ZfDeals\Mapper\Deal' => function ($sm) {
10             return new \ZfDeals\Mapper\Deal(
11                 $sm->get('Zend\Db\Adapter\Adapter')
12             );
13         },
14         'ZfDeals\Mapper\Order' => function ($sm) {
15             return new \ZfDeals\Mapper\Order(
16                 $sm->get('Zend\Db\Adapter\Adapter')
17             );
18         },
19         'ZfDeals\Validator\DealAvailable' => function ($sm) {
20             $validator = new \ZfDeals\Validator\DealActive();
21             $validator->setDealMapper($sm->get('ZfDeals\Mapper\Deal'));
22
23             $validator->setProductMapper(
24                 $sm->get('ZfDeals\Mapper\Product')
25             );
26
27             return $validator;
28         },
29         'ZfDeals\Service\Checkout' => function ($sm) {
30             $srv = new \ZfDeals\Service\Checkout();
31         }
```

```

32         $srv->setDealAvailable(
33             $sm->get('ZfDeals\Validator\DealAvailable')
34         );
35
36         $srv->setProductMapper($sm->get('ZfDeals\Mapper\Product'));
37         $srv->setOrderMapper($sm->get('ZfDeals\Mapper\Order'));
38         $srv->setDealMapper($sm->get('ZfDeals\Mapper\Deal'));
39         return $srv;
40     },
41 ),
42 );

```

Listing 26.48

Die Datei controllers.config.php die Controller-Definitionen:

```

1  <?php
2  return array(
3      'invokables' => array(
4          'ZfDeals\Controller\Admin' => 'ZfDeals\Controller\AdminController',
5      ),
6      'factories' => array(
7          'ZfDeals\Controller\CheckoutForm' => function ($serviceLocator) {
8              $form = new \ZfDeals\Form\Checkout();
9              $ctr = new ZfDeals\Controller\CheckoutFormController($form);
10
11              $productMapper = $serviceLocator
12                  ->getServiceLocator()->get('ZfDeals\Mapper\Product');
13
14              $ctr->setProductMapper($productMapper);
15
16              $validator = $serviceLocator->getServiceLocator()
17                  ->get('ZfDeals\Validator\DealAvailable');
18
19              $ctr->setdealActiveValidator($validator);
20
21              $checkoutService = $serviceLocator
22                  ->getServiceLocator()->get('ZfDeals\Service\Checkout');
23
24              $ctr->setCheckoutService($checkoutService);
25              return $ctr;
26          },
27          'ZfDeals\Controller\DealAddForm' => function ($serviceLocator) {

```



```
28         $form = new ZfDeals\Form\DealAdd();
29         $ctr = new ZfDeals\Controller\DealAddFormController($form);
30
31         $dealMapper = $serviceLocator
32             ->getServiceLocator()->get('ZfDeals\Mapper\Deal');
33
34         $ctr->setDealMapper($dealMapper);
35
36         $productMapper = $serviceLocator
37             ->getServiceLocator()->get('ZfDeals\Mapper\Product');
38
39         $ctr->setProductMapper($productMapper);
40         return $ctr;
41     },
42     'ZfDeals\Controller\ProductAddForm' => function ($serviceLocator) {
43         $form = new \ZfDeals\Form\ProductAdd();
44         $ctr = new ZfDeals\Controller\ProductAddFormController($form);
45
46         $productMapper = $serviceLocator
47             ->getServiceLocator()->get('ZfDeals\Mapper\Product');
48
49         $ctr->setProductMapper($productMapper);
50         return $ctr;
51     },
52     'ZfDeals\Controller\Index' => function ($serviceLocator) {
53         $ctr = new ZfDeals\Controller\IndexController();
54
55         $productMapper = $serviceLocator
56             ->getServiceLocator()
57             ->get('ZfDeals\Mapper\Product');
58
59         $dealMapper = $serviceLocator->
60             getServiceLocator()->
61             get('ZfDeals\Mapper\Deal');
62
63         $ctr->setDealMapper($dealMapper);
64         $ctr->setProductMapper($productMapper);
65         return $ctr;
66     },
67     'ZfDeals\Controller\Order' => function ($serviceLocator) {
68         $ctr = new ZfDeals\Controller\OrderController();
69     }
```

```
70         $ctr->setOrderMapper($serviceLocator
71             ->getServiceLocator()->get('ZfDeals\Mapper\Order'));
72
73         return $ctr;
74     },
75 ),
76 );
```

Listing 26.49

Damit die neuen Konfigurationsdateien vom ModuleManager berücksichtigt werden, müssen die folgenden Methoden in die Module-Klasse eingefügt werden:

```
1  <?php
2  // [...]
3  public function getServiceConfig()
4  {
5      return include __DIR__ . '/config/services.config.php';
6  }
7
8  public function getControllerConfig()
9  {
10     return include __DIR__ . '/config/controllers.config.php';
11 }
12 // [...]
```

Listing 26.50

In der `module.config.php` sind nun nur noch die Routen-Definitionen und die View-Konfiguration enthalten.

Ein besserer Umgang mit Webforms in View-Files

Optimierungswürdig ist auch noch der Umgang mit Formularen in den Views. In allen drei `show.phtml`-Templates des Moduls habe ich in etwa identischen Code:

```

1  <?php
2  $this->form->prepare();
3  echo $this->form()->openTag($this->form);
4  echo $this->formRowTwb($this->form->get('product')->get('id'));
5  echo $this->formRowTwb($this->form->get('product')->get('name'));
6  echo $this->formRowTwb($this->form->get('product')->get('stock'));
7  echo $this->formSubmitTwb($this->form->get('submit'));
8  echo $this->form()->closeTag();

```

Listing 26.51

Das fühlt sich zum Einen nicht wirklich gut an, zum Anderen kann ich keinerlei Änderungen an der Struktur einer Form vornehmen (z.B. ein neues Feld hinzufügen), ohne, dass ich nicht auch das Template anpassen bzw. erweitern müsste. Daher wähle ich hier einen anderen Ansatz: Ich schreibe einen eigenen “View Helper”, der das gesamte Rendering des Formulars kapselt, so dass der Aufruf

```

1  <?php
2  echo $this->renderForm($form);

```

Listing 26.52

sämtliche bisher notwendigen Aufrufe ersetzt. Der “View Helper” `RenderForm`, der zwar sicherlich noch nicht alle “Webform-Eventualitäten” berücksichtigt, für unsere Zwecke aber bereits mehr als ausreichend ist, gestaltet sich wie folgt:

```

1  <?php
2  namespace ZfDeals\View\Helper;
3
4  use Zend\View\Helper\AbstractHelper;
5
6  class RenderForm extends AbstractHelper
7  {
8      public function __invoke($form)
9      {
10         $form->prepare();
11         $html = $this->view->form()->openTag($form) . PHP_EOL;
12         $html .= $this->renderFieldsets($form->getFieldsets());
13         $html .= $this->renderElements($form->getElements());
14         $html .= $this->view->form()->closeTag($form) . PHP_EOL;
15         return $html;
16     }
17
18     private function renderFieldsets($fieldsets)

```

```
19     {
20         $html = '';
21
22         foreach($fieldsets as $fieldset)
23         {
24             if(count($fieldset->getFieldsets()) > 0) {
25                 $html .= $this->renderFieldsets(
26                     $fieldset->getFieldsets()
27                 );
28             }
29
30             $html .= $this->renderElements(
31                 $fieldset->getElements()
32             );
33         }
34
35         return $html;
36     }
37
38     private function renderElements($elements)
39     {
40         $html = '';
41
42         foreach($elements as $element) {
43             $html .= $this->renderElement($element);
44         }
45
46         return $html;
47     }
48
49     private function renderElement($element)
50     {
51         if($element->getAttribute('type') == 'submit') {
52             return $this->view->formSubmitTwb($element) . PHP_EOL;
53         } else {
54             return $this->view->formRow($element) . PHP_EOL;
55         }
56     }
57 }
```

Listing 26.53

Wichtig ist, dass `RenderForm` auch mit Fieldsets und “Fieldsets in Fieldsets” umgehen kann, denn von denen macht `ZfDeals` ja regen Gebrauch. Die rekursive Logik in `RenderForm` sorgt für die korrekte Darstellung auch bei mehrfach verschachtelten Fieldsets. Damit der Aufruf des “View Helpers” wie oben gezeigt funktioniert, muss er zuvor beim `HelperPluginManager` registriert werden. Ich verwende dazu eine eigene Konfigurationsdatei `viewhelper.config.php`

```
1 <?php
2 return array(
3     'invokables' => array(
4         'renderForm' => 'ZfDeals\View\Helper\RenderForm'
5     )
6 );
```

Listing 26.54

und binde Sie über die Methode `getViewHelperConfig()` in der `Module`-Klasse ein:

```
1 <?php
2 // [...]
3 public function getViewHelperConfig()
4 {
5     return include __DIR__ . '/config/viewhelper.config.php';
6 }
7 // [...]
```

Listing 26.55

Nun ist “`ZfDeals`” bereit, um in anderen Anwendungen zum Einsatz zu kommen.

26.8 Ideen für die Weiterentwicklung

Sicherlich hat die Anwendung bis zu diesem Punkt nur einen begrenzten Funktionsumfang. Aber wir haben bereits ein vollständig funktionsfähiges und auf einfache Art und Weise installierbares ZF2-Modul erstellt, das nun sukzessive weiter ausgebaut werden kann. Ein paar technische Dinge gibt es da auch noch, die verbessert werden könnten.

Zend\Di für Dependency Injection

Es ist eine ganze Menge Code erforderlich, um das “Dependency Injection” zu realisieren. Die vielen Factories, die dafür sorgen, dass zum Erzeugungszeitpunkt eines Services dessen Abhängigkeiten aufgelöst werden, werden bereits jetzt etwas unhandlich und die Übersichtlichkeit schwindet. Auch

habe ich das Gefühl, ständig sehr ähnlichen Code zu schreiben und gegen das [DRY-Prinzip](#)²¹ zu verstoßen. Zudem machen die Factories ja eigentlich nichts weiter, als ohne tiefergehende Verarbeitungslogik Objekte zu erzeugen und über “Setter” bereitzustellen. Das ist üblicherweise nicht die Arbeit, die man einer Factory zuschreibt. Hier bietet es sich an, `Zend\Di` zu verwenden.

Doctrine 2 ORM für die Persistenz

Weiterhin fällt auf, dass ein großer Teil des Codes der Anwendung nur dafür da ist, Daten aus der Datenbank zu lesen, oder in ebenjene zu schreiben. Ein ORM-System wie Doctrine 2 kann hier auf einen Schlag dafür sorgen, dass der gesamte Code für die Persistenz überflüssig wird.

Verwendung des Event-Systems

Aktuell gehen wir davon aus, dass das Modul 1-zu-1 so verwendet wird, wie es entwickelt wurde. Was aber, wenn sich der Anwender eines Moduls überlegt, dass es für ihn sinnvoll wäre, wenn nach einer eingegangenen Bestellung eine E-Mail an einen Mitarbeiter gesendet wird. Es empfiehlt daher sich, in der Verarbeitung ein “Business-Event” auszulösen, das es ermöglicht, das Anpassungen und Erweiterungen ermöglicht.

Ein besserer Umgang mit Modul-Assets

Bislang ist der Aufwand, “ZfDeals” in der eigenen Anwendung einzusetzen, noch recht hoch. Das liegt auch daran, dass die “Modul-Assets”, also die CSS- und JS-Dateien sowohl Bilder zunächst vom Modul in den `public`-Bereich der Anwendung kopiert werden müssen. Vermeiden ließe sich das, wenn ein “Asset-Manager” wie [zf2-module-assets](#)²² oder “Assetic” samt den “Glue-Code-Modul” [zf2-assetic-module](#)²³ zum Einsatz kommt, der dafür sorgt, dass “Assets”, die physisch in Modulen lagern, nach Außen hin verfügbar gemacht werden können.



Quellcode-Download

Den Quellcode des Praxisteils, also die Demo-Anwendung samt “ZfDeals”-Modul, findest du zum Download im [Repository “ZfDealsApp”](#)²⁴ und [Repository “ZfDeals”](#)²⁵.

²¹http://de.wikipedia.org/wiki/Don%E2%80%99t_repeat_yourself

²²<https://github.com/albulescu/zf2-module-assets>

²³<https://github.com/widmogrod/zf2-assetic-module>

²⁴<https://github.com/michael-romer/ZfDealsApp>

²⁵<https://github.com/michael-romer/ZfDeals>

27 LAMP-Umgebung einrichten

Als “LAMP-Umgebung” bezeichnet man eine Linux-basierte Laufzeitumgebung für die Ausführung von PHP-Webanwendungen, also einen Rechner für die Entwicklung einer Anwendung oder ein Produktivsystem für den “echten Betrieb”, auf dem ein Webserver mit PHP-Funktion sowie eine Datenbank installiert ist. Klassisch versteht man unter LAMP ein Linux-Betriebssystem, den Apache Webserver, eine MySQL-Datenbank sowie PHP - daher auch die Bezeichnung. Heute versteht man unter LAMP allerdings eine Linux-basierte Laufzeitumgebung für die Ausführung von PHP-Webanwendungen im weiteren Sinne. Es kann also durchaus auch eine NoSQL-Datenbank anstelle von MySQL zum Einsatz kommen oder etwa ein nginx-Webserver, statt dem Platzhirsch Apache.

Die Installation einer LAMP-Umgebung auf dem eigenen System kann je nach Situation und Anforderung durchaus ein zeitraubendes Vorhaben sein. Im Folgenden beschreibe ich Wege, die sehr schnell zu einer einsatzbereiten LAMP-Umgebung führen. Die angenehmste Variante ist die, sich eine virtuelle Maschine erzeugen zu lassen, die alle notwendigen Server-Komponenten enthält.

27.1 Virtuelle Maschine generieren lassen

Um die ersten Schritte mit dem Framework so einfach wie möglich zu gestalten, habe ich eine [Code-Bibliothek¹](#) zusammengestellt, die es auf einfache Art und Weise ermöglicht, automatisch eine virtuelle Maschine auf Basis von “Virtual Box” zu erzeugen, in der die Anwendung ohne weiteres zutun sofort lauffähig ist.

Zunächst wird wie im Kapitel “Hallo, Zend Framework 2!” beschrieben die “ZendSkeletonApplication” heruntergeladen. Dank des Einsatzes von “Composer” ist es sehr leicht, die zusätzliche Code-Bibliothek hinzuzufügen. Dazu wird die `composer.json` im Application-Root im Abschnitt `require` um folgenden Eintrag ergänzt:

1 `"zfb/zfb-vm": "dev-master"`

Die `composer.json` stellt sich nun also wie folgt dar:

¹<https://github.com/michael-romer/zfb2-vm>

```
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": "2.*",
13        "zfb/zfb-vm": "dev-master"
14    }
15 }
```

Auf der Kommandozeile muss nun im Application-Root noch das folgende Kommando ausgeführt werden, das für den Download der Bibliothek sorgt:

```
1 $ php composer.phar update
```

Damit die Erzeugung der virtuellen Maschine auf Basis der Code-Bibliothek funktioniert, müssen nun auf dem Host-System einmalig noch einige Tools installiert werden. Zunächst muss [Virtual Box](#)², dann [Ruby](#)³ (unter Windows am einfachsten über den [Ruby-Installer](#)⁴) und dann noch [Vagrant](#)⁵ heruntergeladen und installiert werden.

Dann die Datei `Vagrantfile.dist` aus dem Verzeichnis `vendor/zfb/zfb-vm` in das Application-Root kopieren und in `Vagrantfile` umbenennen, eine Shell öffnen, in das Application-Root wechseln und die folgenden Kommandos ausführen:

```
1 $ vagrant box add precise64 http://files.vagrantup.com/precise64.box
2 $ vagrant up
```

Eine funktionierende Internet-Verbindung vorausgesetzt, dauert es nun einige Zeit, bis die VM fertig erzeugt ist. Sobald erledigt, kann im Browser über die URL `http://localhost:8080` die Anwendung aufgerufen werden.

Nach getaner Arbeit wird die VM über den folgenden Aufruf im Application-Root in den Ruhezustand versetzt (`http://localhost:8080` ist dann nicht mehr erreichbar):

²<https://www.virtualbox.org/>

³<http://www.ruby-lang.org/en/downloads/>

⁴<http://rubyinstaller.org/>

⁵<http://vagrantup.com/>


```
1 $ vagrant suspend
```

Nun verbraucht sie keine Systemressourcen (CPU, RAM) des Host-Systems mehr, abgesehen von rund 800 MB Plattenplatz. Dafür kann das nächste Mal, wenn man die VM für die Arbeit an der eigenen Zend Framework 2 Anwendung benötigt, die VM innerhalb weniger Sekunden durch den folgenden Aufruf wieder in Einsatzbereitschaft versetzen:

```
1 $ vagrant resume
```