Dependency Injection und Service Locators im Einsatz

Mach dich frei!

Endlich ist es da. Nach fünf Jahren Entwicklung wurde die erste stabile Version von Zend Framework 2 veröffentlicht. Auch wenn viele der Komponenten noch vom Vorgänger bekannt sind, dürfte den meisten Entwicklern schnell klar werden, dass sich etwas grundlegend geändert hat.

von Andreas Baumgart

Die von Altmeister Martin Fowler propagierten und zunächst hauptsächlich in der Java- und .NET-Welt verbreiteten Entwurfsmuster Dependency Injection und Service Locators [1] sind nun das Mittel der Wahl, wenn es um die Verwaltung von Abhängigkeiten geht. Und weil das flexible Spiel mit Komponenten und Plug-ins eine der Stärken des Frameworks ist, werden diese bis vor Kurzem unter PHP-Entwicklern noch recht unbekannten Entwurfsmuster nun zu einem zentralen Baustein für das Design von Zend-Framework-Anwendungen.

Das Problem mit der Abhängigkeit

Ein Großteil der strukturellen Änderungen, die Zend Framework 2 erfahren hat, lässt sich auf eine ganz grundlegende Fragestellung zurückführen. Sie ist eigentlich essenziell für objektorientierte Entwicklung, wird doch erst seit einiger Zeit in der PHP-Welt diskutiert: Wie erfüllt man idealerweise die Abhängigkeiten einer Klasse von anderen Klassen, ohne dabei die Erweiterbarkeit und Portabilität zu blockieren? Wann sich diese Frage überhaupt stellt und warum die Antwort darauf letztendlich eine wichtige Designentscheidung in einer Softwarearchitektur ist, wird im Folgenden anhand eines einfachen Beispiels (Listing 1) diskutiert.

In einer MVC-Webapplikation sollen von einem Controller Aufgaben entgegengenommen werden, die in einer Warteschlange gespeichert und von einem Crongesteuerten Skript sequenziell im Hintergrund abgearbeitet werden.

Die Implementierung der Warteschlange ist relativ einfach gehalten: Die Jobs (auf deren genaue Struktur hier nicht weiter eingegangen wird) werden von der Klasse *Queue* in einem Array verwaltet, mit der Methode *append()* lässt sich ein weiterer Job hinten anfügen, mit *next()* der nächste Job abholen und gleichzeitig vom Stapel entfernen. Um den Status über mehrere Aufrufe hinweg speichern zu können, wird eine ebenfalls recht einfach gehaltene Klasse *FileStorage* verwendet, die PHP-Arrays serialisiert speichern und wieder einlesen kann. Die nötige *FileStorage*-Instanz wird direkt im Konstruktor von *Queue* erstellt. Verwendet wird die Warteschlange in diesem Beispiel in der Action-Methode *createJobAction()* des Controllers *QueueController*.

Drei Arten der Dependency Injection

In PHP kann zwischen drei Arten unterschieden werden, wie Abhängigkeiten in ein Objekt eingespeist werden können. Bei der „Constructor Injection" werden die benötigten Objekte einfach als Parameter des Konstruktors angegeben. Type Hints können einem mit Reflection arbeitendem DI-Container (siehe unten) dabei helfen, die Abhängigkeiten selbst zu ermitteln und ggf. automatisch zu instanzieren. Die Setter-Injection verfährt ähnlich, erlaubt jedoch die Übergabe der Parameter über eigene Mutatoren. Die letzte Variante bildet die Interface-Injection, bei der für jede Abhängigkeit ein Interface bereitgestellt wird, das eine Methode zur „Injection" fordert.

Listing 1: Implementierung einer Warteschlange

```
<?php
// MyModule/src/MyModule/Queue/Queue.php
namespace MyModule\Queue;

class Queue
{
  protected $storage;

  public function __construct()
  {
    $this->storage = new FileStorage();
  }

  public function next()
  {
    $jobs = $this->storage->read();
    $res = array_pop($jobs);
    $this->storage->write($jobs);
    return $res;
  }

  public function append($job)
  {
    $jobs = $this->storage->read();
    $jobs[] = $job;
    $this->storage->write($jobs);
  }
}

// MyModule/src/MyModule/Queue/FileStorage.php
namespace MyModule\Queue;

class FileStorage
{
  protected $filename;

  public function __construct()
  {
    $this->filename = dirname(__FILE__) . '/../../tmp/queue';
  }

  public function read()
  {
    $data = @file_get_contents($this->filename);
    if (false === $data) {
      $data = array();
    } else {
      $data = unserialize($data);
    }
    return $data;
  }

  public function write(array $jobs)
  {
    file_put_contents($this->filename, serialize($jobs));
  }
}

// MyModule/src/MyModule/Controller/QueueController.php
namespace MyModule\Controller;

use \Zend\Mvc\Controller\AbstractActionController;

class QueueController extends AbstractActionController {

  public function createJobAction()
  {
    $queue = new Queue\Queue();

    $job = $this->processUserInput();
    $queue->appendJob();

    return array('job' => $job);

  }
  // ...
}
```

Die genaue Implementierung der ominösen Methode *processUserInput()* ist im Rahmen dieses Artikels belanglos und wird daher der Fantasie des Lesers überlassen. Wir stellen uns einfach vor, dass darin auf magische Weise die Benutzereingaben in Aufgabenbeschreibungen übersetzt werden, die dann später von dem Cron-Job bearbeitet werden sollen.

Da *Queue* so eine praktische Klasse ist, soll sie natürlich auch noch in vielen weiteren Projekten verwendet werden. Einige davon erfordern jedoch, dass die Daten in Memcache oder in einer relationalen Datenbank gehalten werden. Doch weil *FileStorage* direkt im Konstruktor instanziert wird, ist es nicht möglich, die Implementierung einfach auszutauschen, ohne die Klasse *Queue* anzupassen. Das ist zum einen ein Verstoß gegen das Open Close Principle [2] und eigentlich müsste es für *Queue* völlig irrelevant sein, wie genau die Daten gespeichert werden. Benötigt wird lediglich die Information, dass es eine (beliebige) Implementierung gibt, um ein Array mittels der Methoden *write()* zu persistieren und mit *read()* wieder zu einzulesen.

Dependency Injection

Eine mögliche Auflösung dieses Problems bietet das Entwurfsmuster Dependency Injection (DI). Klassen, die nach diesem Muster entworfen werden, vermeiden,

Listing 2: Die Warteschlange mit Dependency Injection

```
<?php

// MyModule/src/MyModule/Queue/Queue.php
namespace MyModule\Queue;

class Queue
{
  public function __construct(StorageInterface $storage)
  {
    $this->storage = $storage;
  }
  // ...
}

// MyModule/src/MyModule/Queue/StorageInterface.php
namespace MyModule\Queue;

interface StorageInterface
{
  public function read();
  public function write(array $jobs);
}

// MyModule/src/MyModule/Queue/FileStorage.php
namespace MyModule\Queue;

class FileStorage implements StorageInterface
{
  protected $filename;

  public function __construct($filename)
  {
    $this->filename = $filename;
  }
  // ...
}

// MyModule/src/MyModule/Controller/QueueController.php
namespace MyModule\Queue;

use Queue\FileStorage;
use Queue\Queue;

class QueueController extends AbstractActionController {

  public function createJobAction()
  {
    $storage = new FileStorage(APPLICATION_PATH . '/../data/queue.txt');
    $queue = new Queue($storage);

    // ...
  }
  // ...
}
```

Listing 3: Die Warteschlange mit Zend\Di verdrahten

```
<?php

// di-config.php
namespace {
  $di = new \Zend\Di\Di();

  $instanceManager = $di->instanceManager();
  $instanceManager->setParameters('Queue\Queue', array());
  $instanceManager->addAlias('queue', 'Queue\Queue');

  $path = '/pfad/zu/queue.txt';
  $instanceManager->setParameters('Queue\FileStorage', array('filename' =>
                                                              $path));
  $instanceManager->addTypePreference('Queue\StorageInterface', 'Queue\
                                                        FileStorage');

  return $di;
}

// worker.php
namespace {
  $di = include 'di-config.php';
  $queue = $di->get('queue'); // oder: $di->get('Queue\Queue')
  processJob($queue->next());
}
```

harte Abhängigkeiten herzustellen. Stattdessen bieten sie Konfigurationsmöglichkeiten an, die es der aufrufenden Komponente erlauben, die Implementierungen zur Verfügung zu stellen, die der Situation am besten entsprechen.

Prinzipiell gibt es mehrere Möglichkeiten, die Abhängigkeiten in eine Klasse hinein zu bekommen (Kasten: „Drei Arten der Dependency Injection"). Wohl am häufigsten anzutreffen ist die Übergabe der Abhängigkeiten an den Konstruktor oder die Verwendung von Settern.

Listing 2 zeigt eine angepasste Version unserer Warteschlange, die sich das Prinzip von Dependency Injection zunutze macht. Zunächst wurde StorageInterface eingeführt, das eine allgemeine Schnittstelle zum Schreiben und Lesen von Arrays definiert und FileStorage wird nun zu einer Implementierung dieser Schnittstelle.

Weiterhin wurde der Konstruktor von Queue angepasst, sodass nun ein Objekt vom Typ StorageInterface übergeben werden kann, an das später die Persistierung, also die dauerhafte Speicherung der Daten, delegiert wird.

Auch die Pfadangabe der Ausgabedatei für *FileStorage* wurde konsequenterweise in einen Konstruktor-Parameter verwandelt, da es sich dabei letzten Endes auch um eine Abhängigkeit handelt, um deren genaue Ausführung sich *FileStorage* keine weiteren Gedanken zu machen braucht, um seine Funktion zu erfüllen.

Das in Listing 1 und Listing 2 verwendete Beispiel verdrahtet die Abhängigkeiten untereinander direkt in der Controller-Logik, was zunächst auch kein akutes Problem darstellt. Erst wenn eine bestimmte Abhängigkeit an mehreren Stellen innerhalb einer Applikation benötigt wird, entsteht redundanter Code, da jedes Mal, wenn eine Queue instanziiert wird, auch eine Instanz einer Unterklasse von *StorageInterface* erstellt werden muss. Dem DRY-Paradigma folgend ist das natürlich kein haltbarer Zustand.

Um solche wiederkehrende Instanziierungsketten zu vermeiden, kann man sich eines so genannten Dependeny Injectors bedienen, der Instanziierung und Konfiguration von Objekten und deren Abhängigkeiten übernimmt.

Zend Framework 2 stellt einen solchen Injector mit der Klasse *Zend\Di\Di* [3] bereit. Instanzen können mit dem einfachen Aufruf *$di->get('Klassenname')* bezogen werden.

Listing 3 zeigt, wie unsere Warteschlange zusammen mit Zend\Di verwendet werden kann. An einer zentralen Stelle, etwa in der *Module.php* oder einem Konfigurationsskript, wird eine Instanz von *Zend\Di\Di* erstellt und konfiguriert.

Mit *$di->instanceManager()* erhält man Zugriff auf den *InstanceManager* von *Di*, der die instanziierten Objekte und ihre Konfiguration verwaltet. Die Parameter, die bereits bekannt sind, werden dem *InstanzManager* gleich mitgeteilt. In dem betrachteten Fall ist das der Pfad der Datei, in der die Queue-Daten der Klasse *FileStorage* gespeichert werden. Außerdem wird festgelegt, welche konkrete Implementierung von *StorageInterface*, nämlich *FileStorage*, verwendet werden soll. Wenn später einmal MemcacheStorage oder DbStorage implementiert wird, wäre lediglich eine Anpassung an dieser Stelle nötig, um die Implementierung zu wechseln. Die Klasse *Queue* selbst bliebe davon unberührt. Da Klassennamen inklusive Namespaces in langen Ketten zur Unleserlichkeit tendieren, bietet *InstanceManager* mittels *addAlias()* auch die Möglichkeit, Aliase für bestimmte Klassen zu definieren.

An anderer Stelle (in unserem Beispiel in dem Skript, das mit der Abarbeitung der Jobs betraut ist) kann man nun über den Injector die konfigurierte Queue beziehen.

Um zu ermitteln, welche Abhängigkeiten die zu instanziierenden Objekte besitzen, kennt *Zend\Di\Di* mehrere Strategien, die sich alle von *Zend\Di\Definition* ableiten. Standardmäßig wird auf die *RuntimeDefinition* zurückgegriffen, die sich des *Reflection*-API bedient, um die Struktur der zu instanziierenden Klasse zu analysieren und Rückschlüsse über die benötigten Parameter zu ziehen.

Der Einsatz von *Reflection* ist während der Entwicklung natürlich sehr bequem, jedoch alles andere als performant. Sollte sich die Auflösung der Abhängigkeiten als Flaschenhals erweisen, so bietet Zend\Di die Möglichkeit, auf weniger dynamische Definition auszuweichen. Die schnellste Lösung bietet *ArrayDefinition*, die, wie der Name schon vermuten lässt, über ein einfaches PHP-Array konfiguriert werden kann.

Wer einen Mittelweg sucht, wird bei der *CompilerDefinition* fündig, die eigentlich eher den Namen *DefinitionCompiler* verdient hätte. Denn *CompilerDefinition* kann den bestehenden Quellcode analysieren und daraus etwa eine *ArrayDefinition* erstellen (Listing 4).

Zusammen mit einem einfachen Caching-Mechanismus und der *RuntimeDefinition* als Fallback kann so

Listing 4:Verwendung von CompilerDefinition

```
<?php

namespace {
  use \Zend\Di\Definition\ArrayDefinition;
  use \Zend\Di\Definition\CompilerDefinition;

  $queueDefinitionPath = '/pfad/zu/queue-definition.php';

  if (!file_exists($queueDefinitionPath)) {
    $compiler = new CompilerDefinition();
    $compiler->addDirectory(__DIR__);
    $compiler->compile();
    $content = '<?php return ' . var_export($compiler->toArrayDefinition()->toArray(),
                                                                          true) . ';';
    file_put_contents($queueDefinitionPath, $content);
  }

  $di = include 'di-config.php';
  $di->definitions()->addDefinition(new ArrayDefinition(include $queueDefinitionPath));
}
```

Listing 5: ArrayDefinition von Queue\FileStorage

```
<?php

namespace {
  $definitionConfig = array(
    array(
      'Queue\FileStorage' => array(
        'supertypes' => array('Queue\StorageInterface'),
        'instantiator' => '__construct',
        'methods' => array('__construct' => true,),
        'parameters' => array(
          '__construct' => array(
            'Queue\FileStorage::__construct:0' => array(
              'filename', null, true,
            ),
          ),
        ),
      ),
    ),
  );

  $di = include 'di-config.php';
  $di->definitions()->addDefinition($definitionConfig);
}
```

ein dynamisches und dennoch performantes System zur Auflösung von Abhängigkeiten für eine *Zend\Di\Di*-Instanz erstellt werden (Listing 5).

Service Locators

Zend\Di war in der ursprünglichen Fassung von Zend Framework 2 als Standardmechanismus zum Verwalten von Abhängigkeiten der MVC-Komponenten gedacht und der Controller wurde einfach über den Dependency Injector konfiguriert. Im Laufe der Entwicklung stellte sich Zend\Di jedoch als zu wenig performant heraus, um an dieser zentralen Stelle verwendet werden zu können. Daraufhin wurde Zend\ServiceManager, das durch den kompletten Verzicht auf Reflection wesentlich schneller arbeiten konnte, zum präferierten Ansatz [4] und löste Zend\Di innerhalb der MVC-Komponenten weitgehend ab.

Zend\ServiceManager stellt eine Implementierung des Entwurfsmusters „Service Locator“ dar, das zunächst gewisse Ähnlichkeiten zu einem Dependency Injector aufweist. In beiden Ansätzen wird eine zentrale Verwaltung für Instanzen und ihre Abhängigkeiten geschaffen. Während bei Dependency Injection jedoch die Abhängigkeiten von außen eingespeist werden (womit es dem Prinzip von „Inversion Of Control“ folgt), muss sich die konsumierende Komponente bei Service Locators um die Beschaffung der von ihm benötigten Services selbst aktiv kümmern. Service Locators, so wie sie in Zend Framework 2 verwendet werden, stellen somit

Listing 6: Warteschlange mit Zend\ServiceManager

```
<?php

// MyModule/Module.php
namespace MyModule;

use Zend\ModuleManager\Feature\ServiceProviderInterface;
use Zend\ServiceManager\ServiceManager;

class Module implements ServiceProviderInterface
{
  public function getServiceConfig()
  {
    return array(
      'factories' => array(
        'Queue\Queue' => function(ServiceManager $sm) {
          return new \Queue\Queue($sm);
        },
        'Queue\StorageInterface' => function(ServiceManager $sm) {
          return new \Queue\FileStorage(__DIR__ . '/queue.txt');
        }
      ),
      'aliases' => array(
        'queue' => 'Queue\Queue',
        'queue_storage' => 'Queue\StorageInterface',
      ),
    );
  }
  // ...
}

// MyModule/src/MyModule/Controller/QueueController.php
class QueueController extends AbstractActionController
{
  public function createJobAction()
  {
    $queue = $this->getServiceLocator()->get('queue');
    // ...
  }
  // ...
}

// MyModule/src/MyModule/Queue/Queue.php
class Queue
{
  public function __construct(ServiceManager $sl) {
    $this->storage = $sl->get('queue_storage');
  }
  // ...
}
```

eine Singleton Factory dar, die für die Verteilung, nicht aber für die Zuweisung der Services bzw. Abhängigkeiten zuständig ist.

Ein bedeutender Unterschied zwischen den beiden Mustern ist die unbedingte Abhängigkeit zum Service Locator, die bei allen konsumierenden Komponenten gegeben ist. Mittels Dependency Injection entwickelte Komponenten können praktisch beliebig in andere Umgebungen übertragen werden – ein zentraler Injector ist optional. Nicht so bei Service Locators, die immer als Abhängigkeit bestehen bleiben und daher ebenfalls portiert werden müssen.

Wie der Ansatz der Service Locators in Zend Framework 2 umgesetzt wurde, zeigt Listing 6, das unsere Warteschlange auf die Modulstruktur von Zend Framework 2 überträgt.

Zunächst wird *Module.php* so abgeändert, dass es *ServiceProviderInterface* implementiert und die Methode *getServiceConfig()* anbietet. Der explizite Verweis auf *ServiceProviderInterface* ist hier im Prinzip optional, da Zend Framework lediglich auf die Existenz der Methode prüft. Doch um des guten Stils willen sollte man das verwendete Interface angeben.

Die Methode *getServiceConfig()* liefert ein assoziatives Array zurück, das die Servicekonfiguration für *Zend\ServiceManager\ServiceManager* enthält und das mit denen der anderen Module zusammengeführt wird.

Die einfachste Möglichkeit, einen Service unter die Obhut des Service Manager zu stellen, besteht darin, im Konfigurations-Array unter dem Schlüssel *invokables* eine Zuordnung des gewünschten Servicenamens zu der zu instanziierenden Klasse herzustellen. Voraussetzung dafür ist natürlich, dass der Konstruktor der Klasse keine Parameter erwartet und auch sonst keine externe Konfiguration der Instanz notwendig ist.

Muss ein Service zunächst konfiguriert werden oder ist auf bestimmte Konstruktorparameter angewiesen, empfiehlt es sich, auf eine Factory zurückzugreifen. Diese werden unter dem Schlüssel *factories* dem Servicenamen zugeordnet und können als Wert beliebige Callables enthalten, *Zend\ServiceManager\FactoryInterface* implementieren oder, unter den selben Bedingungen wie *invokables*, den Klassennamen einer Factory enthalten. Für leichtgewichtige Services kann auch gleich eine konkrete Instanz unter *services* hinterlegt werden. Hierbei geht jedoch der Vorteil einer ressourcenschonenden Initialisierung bei Bedarf („Lazy Loading“) verloren.

Unsere Warteschlange besitzt nun eine Abhängigkeit zum *ServiceManager*, die beim Konstruktoraufruf angegeben werden muss. Und *FileStorage*, die Implementierung, die vom *StorageManager* bei Anfrage eines *StorageInterface* zurück liefern soll, muss mit einem Dateinamen konfiguriert werden. Also entscheiden wir uns für die Initialisierung per Factory in Form von Closures. Werden die Service- und Klassennamen zu lang oder zu konkret, kann man, wie schon bei *Zend\Di\Di*, unter *aliases* knackigere Bezeichnungen hinterlegen. Im Beispiel wird aus *Queue\Queue* ein etwas kürzeres *queue*.

Da *QueueController* von *AbstractController* ableitetet, der wiederum *ServiceLocatorAwareInterface* implementiert, bekommt die *Action*-Methode über *getServiceManager()* den konfigurierten Service Locator zurück, über den schließlich eine Instanz von *Queue* zur weiteren Verwendung bezogen werden kann.

Wie bereits erwähnt, besitzt *Queue* nun keine direkte Abhängigkeit mehr zu *StorageInterface*, dafür muss sie nun mit einem *ServiceManager*-Objekt initialisiert werden, von dem alle benötigten Services, also auch der zum Persistieren der Warteschlange abgeleitet werden. Indirekt besteht diese Abhängigkeit freilich immer noch, denn auch wenn es keine quasi-typsichere Übergabe der Konfiguration (etwa mittels Typehints) gibt, so muss *Queue* immer noch wissen, wie die Schnittstelle des bezogenen Services aussieht. Dass dies nicht explizit geschieht, kann man durchaus als kleinen Schönheitsfehler der Methodik betrachten.

Fazit

Die architektonische Entscheidung für Service Locators oder Dependency Injection sollte wohl überlegt sein: Nach Dependency Injection entwickelte Komponenten lassen sich zwar ohne Weiteres in eine Architektur mit Service Locator integrieren. Der umgekehrte Weg ist jedoch ungleich schwerer, da die Komponenten fest mit dem Service Locator verzahnt sind.

Wer ausdrucksstarken Code hoch bewertet und bereit ist, den Preis zu bezahlen, den ein vergleichsweise langsamer Injector mit sich bringt, der dürfte mit einem zentralen Injector gut beraten sein. Doch wenn Performanz im Vordergrund steht und weniger explizite Abhängigkeiten nicht abschrecken, wird man eher mit Service Locators glücklich werden.

Zend hat sich bei den MVC-Komponenten letztlich für *Zend\ServiceManager* und damit für Service Locators entschieden. Entwickler von MVC-Anwendungen werden in Zukunft um Service Locators also keinen Bogen machen können.

Andreas Baumgart ist freiberuflicher Konzepter und Softwareentwickler mit Fokus auf Web- und Mobile-Applikationen für Endgeräte aller Art. Sie erreichen ihn unter ab@polycast.de.

Links & Literatur

[1] http://martinfowler.com/articles/injection.html#FormsOfDependencyInjection

[2] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[3] http://framework.zend.com/manual/2.0/en/modules/zend.di.introduction.html

[4] http://framework.zend.com/wiki/display/ZFDEV2/RFC+-+ServiceLocator