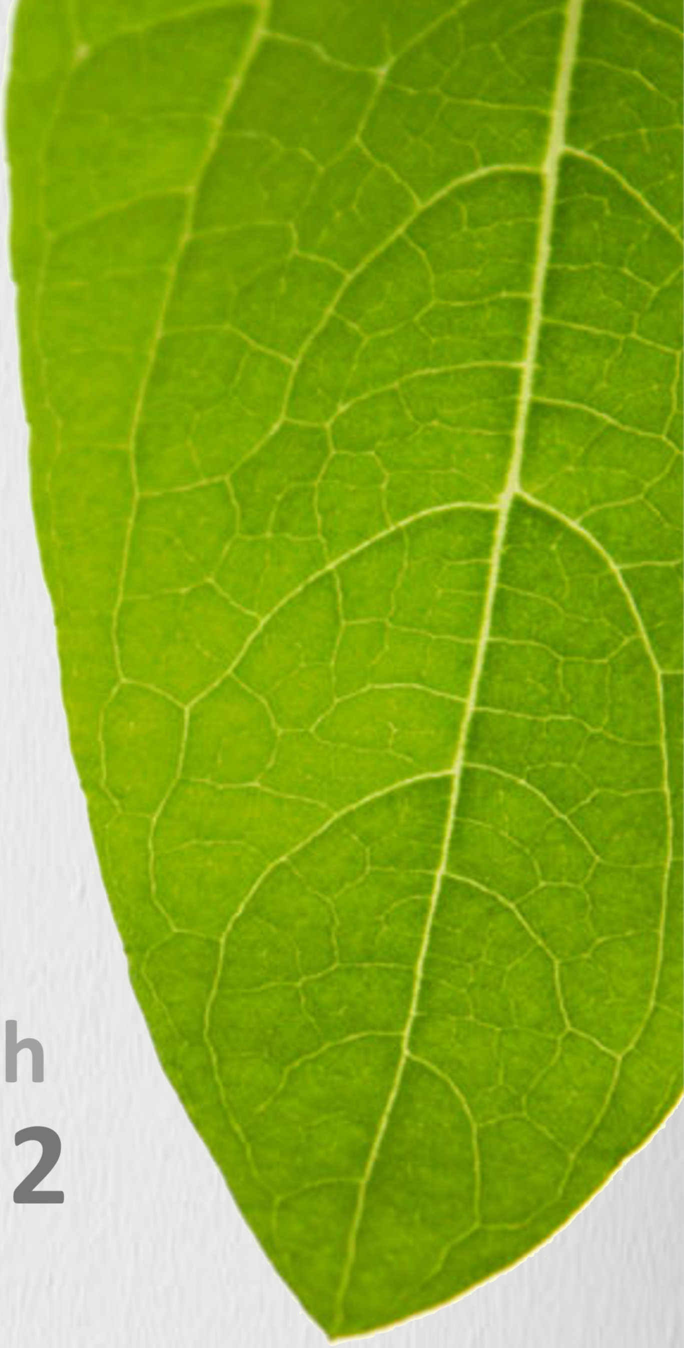


Web Development with Zend Framework 2

Concepts, Techniques and Practical Solutions

Michael Romer

Web Development with Zend Framework 2

Concepts, Techniques and Practical Solutions

Michael Romer

Contents

About the book

Early Access Edition

If you are reading this section, you are holding the “early access edition” of this book in your hands. “Early access” means that you can already begin reading the book while the subsequent chapters are still being written. They will be made available to you as soon as they are finished. Thanks to the lean publishing concept[1], you are also now much more up to date than you have ever been before. The contents of this book refer to Version 2.0 of Framework, they will very probably also unconditionally valid to later versions. It is very possible that you might find a number of spelling errors or also a bug or two in the code examples in the early access edition. At the present time no one has supported me with proofreading or editing in order to reduce the number of errors in this book to a minimum. Naturally, I attempt to be work as carefully as possible, but I cannot guarantee absolute freedom from errors at this time. Please bear with me. If you would like to help me improve this book, feel free to participate in the book’s online community and let me know about your ideas or any issues you discover within in the book. I would greatly appreciate it.

The book’s online community

Found a bug within the book? Want to talk about the book contents or Zend Framework 2 in general? Please feel free to join the book’s online community, the Google-Group “Web Development with Zend Framework 2 Book”[2]. You will need a Google Account which may be set up free of charge.

Important notice for Amazon customers

If you bought this book at Amazon, it is currently somewhat more difficult to ensure that you automatically receive all of its updates as they are published. To avoid problems, please send a short email with “Updates ZF2 book” in the subject line to zf2buch@michael-romer.de. In this manner, you can be certain that you will really always have the latest version of the book. As thanks for your efforts you will also additionally receive the PDF and EPUB versions of the book.

[1]http://leanpub.com/
[2]https://groups.google.com/forum/#!forum/web-development-with-zend-framework-2-book

Introduction

Zend Framework 2 is an open source Framework for the development of professional web applications for PHP Version 5.3. and higher. It is operated on the server and is primarily used to produce dynamic web contents or conduct transactions such as purchasing a product in an online shop. In this context, Zend Framework 2 can support the development of any type of (web)application because it provides universally applicable solutions that can be used in e commerce, content, community or SaaS applications equally well. Zend Framework is essentially being developed by the Zend Company, which also gave Framework its name and provides it with a sold (financial) basis, not least also because Zend itself is also involved in the development of the PHP programming language. However, in addition to Zend, a number of prestigious companies also support Zend Framework and are interested in its long-term success. Among them are also Microsoft and IBM. All of the above provides security for the selection of Zend Framework and also makes its selection a good decision as the basis for one's own application from an economic point of view.

Zend Framework in the stable Version 1.0 first appeared on the scene on 30/06/2007 and has since made a lasting impression on the face of web development with PHP One can truly say that programming with PHP also first became really acceptable for applications critical to company with Zend Framework. If one previously put one's trust in complex J2E (Java) applications in corporate contexts, in the meantime PHP and Zend Framework are gladly chosen with a clear conscience because this combination indeed provides a balanced ratio of lightness and professionalism that practically no other platform can achieve.

If one takes the first preview releases of Zend Framework 1 into consideration, today Framework is already more than 6 years old. Even if much has been achieved in the innumerable releases from the first preview up to the current version 1.11, a number of urgently needed improvements and extensions could no longer be implemented on the old code base; this justifies a new major release, which is for the first time no longer compatible to earlier versions. It is time for a new beginning.

Zend Framework 2 marks the next milestone in the evolution of the PHP web frameworks, but also of PHP itself, for as one has recently seen on the examples of Java and Ruby, a good programming language alone is not enough to also be really successful on a broader front. Only with frameworks—such as Struts, Spring, Rails or indeed also Zend Framework—that are based on the programming language and also significantly reduce the initial development effort but also, long-term maintenance effort of an application, does a web development platform really become established.

For whom is this book?

It is a challenge to write a technical book which finds a balance between theory and practice and allows both novices and professionals to get the best out of the book. I gladly accepted this task, but I left myself an escape hatch. If I have the feeling that we are getting lost in a forest of details that cannot be explained in greater detail at the respective location or even entirely in this book, I refer the reader to passages later on in the book or to secondary literature sources.

A further challenge is whether or not my readers have any previous knowledge of Zend Framework. A developer who has only just begun to work with Version 2 of Zend Framework requires different information in some places than an "old hand", quickly finds his or her way around the many corners in Version 2 because he or she is already familiar with ideas and concepts from Version 1. For all those who are familiar with Version 1, I will frequently refer to the predecessor of Version 2 at appropriate places—whenever I consider it necessary—without going into excessive detail. That might also perhaps help novices, because in this manner they would get a better feeling for why Version 2 is necessary. This book should be helpful for both novices and advanced learners of Zend Framework.

I presume that you have basic knowledge of PHP. You do not have to be a PHP expert, particularly because many "native" PHP functions even became obsolete when one used Framework, for example, Session Management, which maps in an object-oriented manner and in this manner thankfully abstracts some of the low-level functionality. Hence, if you are accustomed to PHP syntax, have a basic understanding of the operating principles of PHP applications and are familiar with the common functions of the language core, you are well prepared. If necessary, you will also have to use a PHP handbook.

You & I

Hi, I am Michael. I hope that you won't object if I occasionally use contractions and the less formal "you" instead of "one" in my explanations. That makes writing it easier form me and ensures a less formal atmosphere.

Structure of this book

This book is not meant to be a compendium, but rather a pragmatic and practice-oriented introduction to the basic concepts and practical work with Zend Framework 2. From a certain point in one's progress as a developer onward, the official documentation serves as a compendium for experienced developers, but it is not really appropriate for use while you are becoming familiar with the subject; instead, it serves as an (indispensible) reference work for further detailed questions after one has achieved the required basic understanding of the program. And exactly that is this book's objective.

It is structured such that you can read it from the beginning to the end, and that is what you should do. We will begin with an overview of Framework and will first look at the essential concepts and ideas which make up the essence of Framework and also differentiate it from its predecessors. On the way, we will repeatedly also look to the left and to the right and thus become familiar with some framework conditions, for example how Framework was really developed. Then we will go into more detail and elucidate Framework's most important components and relationships; then take a look at how a HTTP request is processed and write our first bit of code. This is followed by an excursion into the Framework environment. We will examine the most important modules, which have been made available by third parties and, for example, make the realisation of a user administration of one's own obsolete. A large part of the magic of the new Framework version is the result of the module concept, and modules can—as we will see later—greatly accelerate and simplify the development of your own applications. It is to be expected that in a short time a large number of high-quality modules will be available for Zend Framework 2.

Last but not least, there is an additional, intensive practice section in the form of a "developer's diary". We will develop a web application together, which is also intended to really be used subsequently by a business enterprise; for this reason it is better if we really try hard. At the latest, we will begin doing real "hands-on" work at this point.

Repetitions

You will soon realise that I frequently explain contexts several times in this book This can be due to two things: 1) that I lost track of what I had already said and what I had not (:-D) or 2) that I consciously intended to do it. For even the old Romans knew that: Repetitio mater studiorum est – Repetition is the mother of learning. But also the fact that the respective contexts are in a different context in each case and thus discussed from another perspective is also conducive to comprehension. Thus, if you happen to find a place in the book where you think, "I already know all that!", just be glad that you have learned so much and continue reading or just skip the respective passage.

How you can best work with this book

Programming (or handling a programming framework) is best learned when you become active yourself. Indeed, particularly the first chapter up to the practice section is already helpful even without an opened IDE, but you will have the greatest possible learning success if you reproduce some of the lines of code or write some yourself. In the ideal case, you have a system with a debugger at hand, with which you can follow Framework's mode of operation step by step. Some knowledge of Git and a GitHub account would not hurt anything either, but are not essential.

Found a bug?

Have you found a bug in the text or code? Please feel free to file a bug ticket in the bug tracker[3]. I am thankful for your feedback and support.

[3]https://github.com/michael-romer/zf2book/issues

Conventions used in this book

Code examples

The listings in this book have Systax Highlighting, wherever possible, but do not always conform to a coding standard; this serves to make them more legible. PHP code is introduced by a `<?php`, and a `// [..]` in the respective listing indicates excluded code fragments.

Many listings have a link to GitHub, where they can be downloaded in the form of a so-called "gist" or simply be adopted with "Copy & Paste". In this manner, code examples can easily be re-enacted on your own system.

Command line

When a command has to be executed on the command line, this is symbolised by a preceding dollar sign ($) . The visual feedback of the command is indicated by a preceding "greater than sign" (>). Example:

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

So, that is enough of the foreword, let's get down to work!

Zend Framework 2 - An overview

Before we begin to immerse ourselves in the details of Framework in the further course of the book, we initially want to get an overview and to elucidate some of the core aspects and thoughts underlying Framework. What are Framework's main characteristics?

How Framework is being developed

Zend Framework 2 is open source. Initially, this means that anyone can examine the source code and use it for his or her own purposes. Framework is being developed under the "New BSD License". Whereas under the original "BSD License" it was still necessary to refer to the use of a library or a framework under "BSD Licence", this is not necessary for Zend Framework 2; the so-called "Marketing Clause" does not exist in the "New BSD License". We do not want to drift off into software licensing law, but do want to say at this point that, with reference to the law, that Framework is making things easy for us because the "New BSD License" indeed belongs to the so-called free software licenses, which have practically no limitations or directives for the use of the code.

Whereas Version 1 still used SVN for the code administration, the project team for the new release decided to administer the Framework code on GitHub[4]. GitHub supports the joint, distributed work on the code base very well, especially since many programmers are now involved in the development and are spread across the globe. Thus, Rob Allen resides in England, whereas Matthew Weier O'Phinney, Project Head for Zend Framework 2 lives in the USA, and Ben Scholzen, in Germany. The above-mentioned programmers are all so-called "code contributors", i.e. they have already made a significant contribution to Framework and thus have a special status in the team. They decisively shape Frameworks structure and design. Some of the programmers are directly employed at Zend, for example Matthew Weier O'Phinney; others are freelancers or have other working conditions which allow them to collaborate on Zend Framework—during or after their working hours. Thus, it is a colourful group of good people. Zend Framework's component-orientation allows the use of so-called "Component Maintainers[5]", who are responsible for a specific Framework component, and they themselves or jointly with other programmers control the fate of a certain component.

The team organises itself primarily via Wiki[6], mailing lists and IRC chats. The "Zend Framework Proposals Process" provides everyone with the possibility of submitting suggestions for the further development of Framework and regular "Bug Hunting Days" help resolve known Framework problems in a focused manner by joining forces. New versions of Framework, with which bug fixes and new features are made available, are published regularly.

[4]https://github.com/zendframework/zf2

[5]http://framework.zend.com/wiki/display/ZFDEV2/Component+Maintainers

[6]http://framework.zend.com/wiki/dashboard.action

The PSR-2 Coding Standard

The code of Framework itself is being developed across all components under consideration of the PSR-2 Coding Standards[7]. The idea of the "PHP Specification Request", "PSR" for short, was inspired by the Java Specification Request[8]. Using this procedure, new Java standards are defined and extensions of the Java programming language or the Java runtime environment are jointly developed and are agreed upon by all manufacturers. This procedure has many advantages for the application developer because it makes the application proper much more portable and manufacturer-independent. Thus, it is possible in a (more or less) simple manner, for example, to change the provider of one's own application server.

The "PHP Specification Requests" are based on a similar idea—they should, in particular, ensure that software components made by different manufacturers and frameworks are compatible to one another and can be used in combination. In contrast to "JSR", the "PSR" procedure is relative new. To date only three specifications have been agreed upon.

- PSR-0: Defines the coherency between PHP namespaces and the organisation of PHP files in a file system in order to make the autoloading of classes, which should also be component- and manufacturer-independent, as simple as possible.
- PSR-1 / PSR-1 basic: A new common coding standard.
- PSR-2: An extension of the PSR-1 Coding Standard.

A primary focus of Zend Framework Version 2, as we will repeatedly see in the course of this book, is on the functional extension of an application by means of reusable and simply integrated modules. Compliance with the PSR standard is a great help in this context.

Known problems

As is the case for every large piece of software, Framework is not completely free of errors. In a consequent manner, github issues[9] is used for tracking .... Thus, if you discover a problem, it is worthwhile to initially look there to see whether the error is already known. If not, you can open a thread there.

Module system

Das module system is the central hub of Version 2. Matthew Weier O'Phinney stated this extremely clearly in the mailing list:

[7]https://github.com/pmjones/fig-standards/blob/psr-1-style-guide/proposed/PSR-2-advanced.md

[8]http://de.wikipedia.org/wiki/Java_Specification_Request

[9]https://github.com/zendframework/zf2/issues?state=open

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

The Framework Module System was completely reworked for Version 2. Even though it was indeed already possible to organise one’s own code in modules, those modules were never really independently usable nor could they be transferred to other applications without having to fit the module into the respective code with a great deal of effort there. The effort of doing this was normally so great that one could just as easily program the function him- or herself The module system of Version 1 was thus restricted to the advantages of a better code organisation within a self-contained application. The result is that we now probably have thousands of implementations for the authentication of a user, or similar functions, which are generally applicable because they are not restricted to a specific application.

One is accustomed to adding functional extensions in the form of plugins or extensions from applications such as WordPress, Drupal or Magento. Even Symfony 2[10]—a popular, alternative web framework— with its bundles has already had a module concept, which makes it, for example, possible to integrate the functionality of a content management system (CMS) into one’s own application without having to program it oneself, for some time. And now Zend Framework has also included this extension option in its new version. Let’s take a concrete example again: An application, which was developed on the basis of Zend Framework, subsequently additionally requires the functionality of a blog. This is a customary requirement because a blog is extremely practical, for example, as an SEO measure. Anyone who has worked with Wordpress & Co. knows how comprehensive the requirements for a modern blog system have meanwhile become. It quickly becomes clear that it is not a good idea to now develop one’s own blogging software.

Instead, it appears appropriate to use one of the available free or commercial blogging systems, which however due to its concept can only be set up in parallel to one’s own application. This has a number of disadvantages: For example, it is not easily possible to use the logging system of one’s own application without further ado; nor is this possible for the caching layer. The blog’s data are located in another database, and if we want to display the last 3 blog post teasers on our applications homepage, we have implement this tie-in via one of the blog’s APIs, possibly over an RSS feed or something similar (or mess around in the database of a third party application …). Naturally, the administrator accounts which allow our employees to administer customer master data do not exist in the blog system, and we should not even mention Single-Sign-On[11]. We have to simulate our corporate design in the blog’s template system because layout, markup and styles are not readily available there. Any future design adaptations will also always have to be reconstructed there. Our application’s build scripts cannot be used for the blog; in contrast, the release processes must be adapted so that we can somehow also include the third party blog system. Thus, with this approach we skid directly into the complexity of Enterprise Application Integration[12] (EAI) and Service Oriented Architecture[13] (SOA), and in this manner we create a colourful bouquet of new problems and challenges for ourselves. If instead a blog module for Zend Framework 2 were

[10]http://symfony.com/

[11]http://de.wikipedia.org/wiki/Single_Sign-on

[12]http://de.wikipedia.org/wiki/Enterprise_Application_Integration

[13]http://de.wikipedia.org/wiki/Serviceorientierte_Architektur

available to use—one which would seamlessly integrate itself in the existing authentication, build & release, caching, logging and design implementations—everything would be much more simple, indeed nearly trivial. And exactly this train of thought is the core idea of the module system.

A Zend Framework module brings everything with it that is required for its operation. This includes not only the appropriate PHP code, but also the HTML templates, CSS, JavaScript code, images, etc., so that a Module is a truly self-contained package. A good module can be readily integrated in a Zend Framework 2 application and … simply runs. The module system thus makes Zend Framework 2 to much more than just a web framework; indeed it goes far beyond this and is really a platform for integrated applications and functions.

In the next chapter we will elaborate on the technical details of the module system and take a look at how it functions internally and how modules are developed because even one's own application is represented in code by a module. Modules are everywhere in the new Zend Framework! When you understand the modules, that's half the battle both for the development of one's own application and for embedding already implemented functions and systems into it.

Further on in the book, we will take a look at the available modules because in addition to the already mentioned functions for user management, there is much more; for example, modules related to Doctrine 2[14], the well-known PHP-ORM[15] system. These module use so-called "glue code", which allows one to easily use this library in a Zend Framework 2 application

Event system

Zend Framework 2 is decisively based on the concept of Event-driven Architecture[16]. This approach builds on the idea that certain activities occur in a system after a specific event have previously taken place. To achieve this, activities register themselves for an alert when the event occurs. If the event occurs, the registered activities take place. Here is an analogy from the real world: When we wait at a bus stop (we have "registered" ourselves for the event of the bus's arrival), and when the bus finally drives up (event occurs), its door opens (activity 1), we buy a ticket (activity 2), search for a vacant seat (activity 3), and the bus starts up (activity 4).

Here is another example: An article offered by eBay is sold. This event triggers a series of activities in the system:

- A confirmation of purchase is sent to the purchaser.
- A confirmation of sale is sent to the seller.
- Sales fees are calculated and charged to the seller's account.
- The article in question is removed from the search index, so that it can no longer be found (it has already been sold).

[14]http://www.doctrine-project.org/
[15]http://de.wikipedia.org/wiki/Objektrelationale_Abbildung
[16]http://en.wikipedia.org/wiki/Event-driven_architecture

If eBay decides at a later point in time to also inform the unsuccessful bidders that the auction has ended (in reality this has already been done), this activity can be added to other activities for this event.

However, EDA is a two-edged sword. On the one hand, one achieves an enormous flexibility in structuring workflows by employing this style of architecture. New activities can be easily added. In this manner, entire procedures can subsequently be easily modified in this manner. Flexibility is the decisive argument. In contrast, there is a certain lack of transparency regarding the things that all really take place in the application when an event occurs. Fundamentally, an activity for an event can be registered more or less anywhere in the application without this connection being visible at the location in code where the event subsequently really occurs. A further challenge is the sequence of the activities. If the seller in the eBay example given above, is also notified of the fees due (which are calculated on the basis of the final price in the auction), this activity must occur after the fees have been calculated (i.e. another activity had to take place). Now, things begin to get complicated. In a nutshell, EDA can result in processes that are difficult to understand. The causes of errors are more difficult to identify, and debugging applications is more complicated.

Despite this, the advantages outweigh the disadvantages by so much, particularly also in connection with Framework's module system, that the programmers decided to use EDA for Zend Framework 2. If one is familiar with the pitfalls, one can easily avoid them.

MVC implementation

Also in Zend Framework 2, the implementation of the MVC patterns is at the focus of Framework. Although Zend Framework 2 also again provides the option of freely using its components and, for example, ignoring `Zend\Mvc`, in practical work one would only do this in exceptional cases. Indeed, in most cases, it is precisely the MVC implementation and the resulting advantageous code structure in one's own application that is often the basis for decisions to use Zend Framework. `Zend\Mvc` structures an application via the logical separation of code components into "models", "views" and "controllers", and in this manner ensures a certain order that is not only beneficial for the application's serviceability and extensibility, but also promotes the reuse of functions.

And this is what `Zend\Mvc` in action looks like: After the `Zend\ModuleManager`, the central unit in the Module System, has prepared all the available Modules for use, read in the configuration, and initialised additional components, the `Zend\Mvc\Router` ensures that a suitable Controller (a class) is instantiated in a Module and the correct action (a method) is invoked in it. In this context, the routing is based on previously configured routes, which represent the mapping between URLs und Controllers or Actions, respectively. The selected controller refers back to the Request Object to further process it; this allows an object-oriented access to the Request Parameters, which is made accessible by the `MvcEvent` object. The latter, in turn, was generated at the start and will be made available at the appropriate locations. In the further course, the controller makes use of the application's model and accesses persistent data, services and business logic. It ultimately generates the "view model", on the basis of which and appropriate HTML templates as well as the use of the so-called “view helper” the result of the invocation is generated. Optionally, the output is now also inserted into a layout and the final result is returned to the calling program. But we will go into that in greater detail later.

Since `Zend\Mvc` extensively uses the Framework Event System, a great many options for influencing the standard course sketched above are provided here. In this manner, for example, access control or an input filter can be implemented before the controller is executed. Before the return of the results, one could ensure that the generated HTML markup is error-free, if necessary, with the aid of HTMLPurifier[17].

The code for the MVC implementation is a completely new development. The MVC implementation in Framework Version 1 was still rather inflexible, whereas the new version is definitely more flexible and ultimately allows the configuration of any arbitrarily adapted workflows. Basically, the procedure sketched out above can also be completely differently structured using `Zend\Mvc` without having to dispense with the use of Zend Framework and the advantages resulting from its use.

Additional components

In addition to `Zend\Mvc`, `Zend\View`, `Zend\ServiceManager`, `Zend\EventManager` and `Zend\ModuleManager`, which jointly comprise the “framework core”, Zend Framework 2 has a number of further components, which we will briefly consider in the following. The majority of these components will also be considered in a detailed manner again in the course of this book. At that time we will deal with each of the respective components more intensively. In contrast to many other frameworks, Zend Framework has always been so conceived that it is also possible to use only selected components, while other components are ignored.

When looking at the list of components, those who have used Zend Framework 1 will notice that some components no longer exist. In particular, the many components for linking up to diverse web services are no longer part of Framework in Version 2.0, but are maintained as independent projects or libraries. Thus, for example, `Zend_Service_Twitter` is no longer a part of the program, but is now administered in the Git account `zendframework`[18] in its own repository. The idea behind this is to sharpen Framework’s profile and to focus its application area. Nor does `Zend_Registry` exist in the new version. Its task is now performed by the `ServiceManager`, which (as we will see in the further course of the book) can also do much more. `Zend_Test` has also been removed. The good news is that essentially—as a result of the loose coupling of the individual objects and services that predominates in Zend Framework 2—no additional functions, other than those already available in PHPUnit[19], are now required for “unit testing”. This is very good news and as we will see in a subsequent chapter and in the practice section of the book, unit testing has become much simpler in the new version.

As standard, the following components are additionally contained in im Zend Framework 2.

[17]http://htmlpurifier.org/
[18]https://github.com/zendframework
[19]https://github.com/sebastianbergmann/phpunit/

- Authentication: Serves to implement a "Login" function, in which the computer checks whether or not a user really is the person whom he claims to be (for example, because he knows the secret password).
- Barcode: Library for generating barcodes.
- Cache & Memory: Generic implementation of a caching systems under consideration of different "Backends", for example "Memcached" or "APC".
- Captcha: Generation of CAPTCHAs[20], for example for use in web forms.
- Code: Tools for the automatic generation of code.
- Config: Aid which handles reading and writing of application configurations in extremely different formats, for example YAML or XML.
- Console: Library for using application functionality, for example controller, via a shell (instead of on the basis of a HTTP requests by a browser).
- Crypt: Functions that handle encryption.
- Db: Library for work with databases (but no ORM system).
- Di: Implementation of a Dependency Injection (DI) Container.
- Dom: Library for server-side work with the DOM.
- Escaper: Aid for output escaping.
- Feed: Generation of RSS and atom feeds.
- File: Aid for working with files.
- Filter: Functions for filtering data, for example in the framework of a web form.
- Form: Library for the PHP-assisted, object-oriented generation of web forms under consideration of "Validators" and "Filters".
- Http: Aid for dealing with the HTTP.
- I18n: Extensive library for the internationalisation of applications, e.g. the output of translated contents.
- InputFilter: Allows the use of filters and validators on received data.
- Json: Tools for the serialisation and deserialisation of JSON data structures.
- Ldap: Library for the linkage of LDAP systems, for example in conjunction with Authentication.
- Loader: Autoloading functions for PHP classes as well as for loading MVC modules.
- Log: Implementation of a generic logging functionality with support of different types of "Log Memories".
- Mail & Mime: Library for sending (multipart-)emails.
- Math: Diverse mathematical aids.
- Navigation: Generation and outputting of web site navigations.
- Paginator: Generation and outputting of "Sheet Navigations", for example in results lists.
- Permissions: Library for rights and role systems.
- ProgressBar: Generation and presentation of progress bars (among others also on the command line)

[20]http://de.wikipedia.org/wiki/CAPTCHA

- `Serializer`: Tools for the serialisation and deserialisation of objects, for example for long-term storage.
- `Server`, `Soap` & `XmlRpc`: Library for the generation of web services, e.g. on the basis of SOAP or XML-RPC. Part of `Soap` is also a helpful SOAP Client implementation.
- `Session`: Administration of user sessions.
- `Stdlib`: Diverse standard functions and objects, for example the implementation of a "Userland PriorityQueue", which, e.g., is used by the `EventManager`.
- `Tag`: Functions for the administration of "Tags" and the generation, for example, of "Tag Clouds" on a website.
- `Text`: Aid that handles the management of strings and scripts. Is used internally, e.g., by `Captcha`.
- `Uri`: Functions for the generation and validation of URIs.
- `Validator`: Extensive library for the syntactic validation of data, for example of entries in forms, for many application cases, among them ISBN, IBAN, email addresses and much more.
- `Version`: Holds information on the used Framework Version and the available, newest version at GitHub in readiness.

Design Patterns: Interface, Factory, Manager, etc.

If one looks through Zend Framework 2 code for a time, one notices that it is crammed full of implementations of so-called Design Patterns[21]. If one is more familiar with "typical PHP code—and that is not intended to be judgemental in any way—it will take awhile before one finds one's way around Zend Framework 2. If one has had much to do with Java, one will feel much more rapidly at home, simply because "design patterns" found their way into the Java world several years earlier or even evolved there, respectively. To simply your access to the material, let's take a look at a few ordinary Zend Framework constructs in the following. Not all of them are really Design Patterns in a strict sense, but we should ignore this fact for the moment. And we should also do the same with the fact that I use serviceable simplifications in my explanations at some places in the book. This is not a comprehensive book on design patterns and the knowledge to be imparted is primarily intended to help the reader develop an understanding for Zend Framework. At this time I should perhaps repeat the following advice: Of course, as an application developer it is not necessary to understand all the mechanics of Zend Framework 2 in detail. Quite the contrary: Framework is meant to reduce the work effort and to make it possible for one to concentrate entirely on the programming of the "business logic" itself. However, it is a great help if one has a fundamental understanding of the connections between the individual Framework components—and even more important: of the basic concepts. Thus, it is worthwhile not to skip this chapter.

[21]http://de.wikipedia.org/wiki/Entwurfsmuster

Interface

Interfaces are an inherent element of object-oriented programming, and PHP has supported them comprehensively since Version 5. Interfaces allow one to decouple invoking code from a concrete implementation. If one always develops one's applications for a defined interface, one can rest assured that at runtime a concrete implementation with stipulated methods and properties will be available. Indeed, one can also use an alternative implementation without having to modify the invoking code.

Listener

A "listener" is a short string of code that is executed as soon as a defined event occurs in an application. Technically speaking, to achieve this, a listener is registered beforehand by a so-called "EventManager" (Attention: risk of confusion with a popular profession) for an event. In this manner, the connection convenes at this location.

ListenerAggregate

A "ListenerAggregate" herds a series of listeners together, for example, in order to register them with an "EventManager". Not much more, but also no less.

Factory

A "Factory" is always used when the instantiation of a specific object is complicated, i.e. when an entire series of manipulations are required to make an object ready for use. For example, Zend Framework uses a factory to instantiate its `ModuleManager` and additionally to register a number of module-relevant listeners.

Service

A "Service" provides access to specific files or functions. The term is extremely general and can have a completely different meaning in each case depending on the context. In the scope of Framework a service is understood to mean an object that is made available by a `ServiceManager` and provides a defined service. Thus, for example, listeners registered in the `ServiceManager` are termed services—just as, e.g., the `ModuleManager`, but also ontroller or "view helper", are.

Manager

A "manager" is an object that manages the administration of a specific type of other object in the system. For example, Doctrine 2[22] has a so-called "EntityManager", which administers "entities", i.e.

[22]http://www.doctrine-project.org/

certain persistent objects (e.g., in a shop offerings, categories, customers, orders, etc.) throughout their entire lifecycle and ensures that thee ntities are read from a database and changes are transparently returned.

Strategy

Behind the Strategy Design Pattern is the idea of swapping out algorithms, which one would otherwise "hard wire" at the respective location in the code, to a class of its own and thus to make it exchangeable. Thus, sorting algorithms, for example, are good candidates for this strategy pattern. The different algorithms, according to which, e.g., a product lists can be sorted (price increasing/decreasing, rating increasing/decreasing, etc.) are not permanently encoded, but instead realised in the form of a class of their own, which all implement a common interface, which specifies a `sort()` method. If one has once implemented this mechanism, any other arbitrary sorting procedure can be realised at a later time and then be added.

Model View Controller (MVC)

The MVC pattern decisively affects the structure of the application code because it logically separates those components from one another, which manage the display (View), the processing of user interaction (Controller) and "business logic" with its objects and services.

Actions

"Actions" are an approach for further structuring the code used for processing user interactions in controllers. They are therefore closely connected with the MVC pattern and also are used in Zend Framework 2.

View Helper

With the aid of "view helpers", code for presentation logic can be encapsulated and reused in a standardised way.

Controller Plugins

By means of "Controller Plugins" frequently used code can be organised for interaction processing and be used in several controllers.

Hello, Zend Framework 2!

Put away all the grey theory—let's take a look at Framework in action

As discussed in the last chapter, `Zend\Mvc` is an independent, optional, but essential component of Framework. The following context always includes `Zend\Mvc`. `Zend\Mvc` also dictates its own application and in a certain manner also the directory and code structure . But that is actually quite practical. If one is already familiar with a Zend Framework 2 application, one can also orient oneself very quickly in other applications that are also based on Framework. Although one does indeed have the freedom to establish a completely different directory and code structure, this would make life unnecessarily difficult, as we will see later. If an application to be created, it is wise to use `Zend\Mvc` from the very beginning. However, if one wants to extend an existing application with functions from Zend Framework 2, it is perhaps appropriate to initially dispense with `Zend\Mvc` completely or to first use it at a later time.

Zend Framework 1 and 2 in parallel

One can also operate Zend Framework 2 in parallel to Version 1 and initially only use Zend Framework 2 intermittently.

Installation

In principle, Zend Framework 2 does not have to be tediously installed. One simply downloads the Code[23], makes it available over a web server with PHP installation and can begin immediately. However, the fact that the Zend Framework 2 Code alone is not enough to be able to actually see a `Zend\Mvc`-based application in action is a challenge because, as we have already mentioned, `Zend\Mvc`, i.e. the components which take over the processing of HTTP requests, is optional and accordingly is also not inherently "wired" for use. One must thus initially personally ensure that `Zend\Mvc` is so equipped with configuration and initialisation logic—the so-called "boilerplate code"—that it can also actually be used. Otherwise, one initially sees … nothing.

To avoid this effort and to make getting started with Version 2 as simple as possible, the so-called "ZendSkeletonApplication" was developed in the course of Framework's development; this serves as a template for one's own project and includes the necessary "boilerplate code", which one would otherwise have to prepare oneself with great effort.

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

ZendSkeletonApplication

The installation of the "ZendSkeletonApplication" is the simplest with help from Git. To take advantage of this, it is first necessary to install Git on one's own computer. On Mac systems and in many Linux distributions, Git is even already preinstalled. For installation on a Windows' system, Git for Windows[24] is available for downloading. The installation under Linux nearly always runs under the respective package manager. After installation and after invocation of

```
$ git --version
```

on the command line, one should see this or a similar "sign of life":

```
> git version 1.7.0.4
```

From here onwards, everything is very easy—change to the directory in which the subdirectory for the application is to be set up and which can later be made available to the web server as "document root", and download the ZendSkeletonApplication .

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

Admittedly, in Git jargon it has to be termed "cloning" and not "downloading". But for the time being, we will ignore that. And by the way, do not be afraid of Git! One does not need any advanced Git knowledge in order to successfully work with this book and Framework. Of course, the reader can also administer his or her own application code in the future even permanently— with Git, but it is not necessary. Therefore, a subversion, CVS or even no system at all can also be subsequently used for code administration without problems.

Downloading the "ZendSkeletonApplication"" is very fast, even for less rapid Internet connections, but one must always have such a connection in any case. The reason for the fast download is the fact that Framework code itself is not downloaded at all; instead only the corresponding boilerplate code for the development of one's own application, which is based on Zend Framework 2, is provided.

Composer

The "ZendSkeletonApplication" uses with Composer[25] another PHT tool, which established itself for the management of dependencies for other code libraries some time ago. The idea behind the composer is as simple as it is ingenious. A configuration file contains a definition of the other code libraries that an application is dependent on and from where the respective library can be obtained. In this case, the application is dependent on Zend Framework 2, as can be seen by looking in the file composer.json in the application root.

[24]http://code.google.com/p/msysgit/
[25]getcomposer.org/

```
 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[26]

In lines 11 and 12, the application's two dependencies are declared. Both PHP 5.3.3 or higher and the current version of Zend Framework 2 are required. The following two invocations ensure that Zend Framework 2 is downloaded and additionally also integrated in the application such that it is immediately utilisable and the corresponding Framework Classes are made available by autoloading.

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

Composer has now downloaded Zend Framework 2 and made it available for the application in the `vendor` directory.

Phar-Archive

A Phar Archive provides the option of making a PHP application available in the form of a single file. If one looks at the Composer-Repository at GitHub[a], it becomes clear that composer does not consist of a single file, as one might think, but that its components are merely bundled in a Phar Archive for distribution of the application.

[a] https://github.com/composer/composer

[26] https://gist.github.com/3820657

Phar Archive and Suhosin

If "Suhosin""[a] is used on a system, the use of Phar must initially be explicitly permitted such that the `suhosin.ini` is extended by the entry `suhosin.executor.include.whitelist=phar`. Otherwise, problems can occur in the execution of the Composer command.

[a]http://www.hardened-php.net/suhosin/

Installation without Git or Composer

If necessary, it is also possible to obtain Framework and the "ZendSkeletonApplication" via a "normal download" or Pyrus (the successor to PEAR). Additional installation information is to be found on the official download site[a].

[a]http://framework.zend.com/downloads

A first sign of life

We have now completed nearly all of the required preparations. Finally, we only have to ensure that the application's `public` directory is configured as Document Root of the web server and can be called up/invoked via the URL 'http://localhost by the browser.

For example, to achieve this, a directive in following exemplary form must be specified in the httpd.conf of Apache:

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

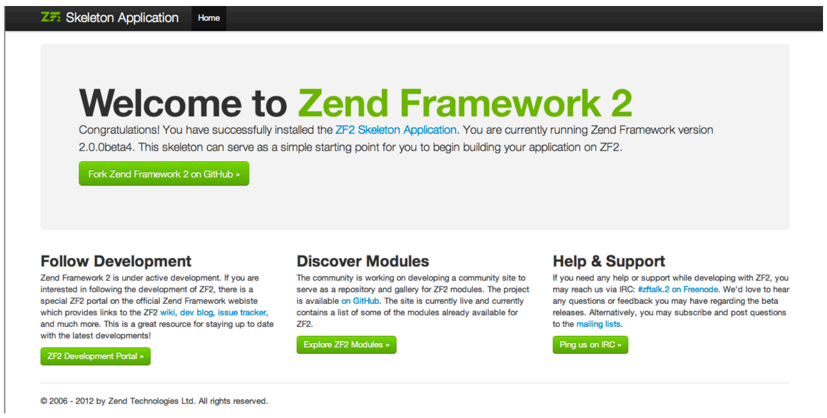
where it is required that the "ZendSkeletonApplication" was downloaded with the following command beforehand:

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

Setting up a PHP runtime environment.

Since the scope of my readers' previous knowledge is probably extremely different, I will not explain exactly how a web server is installed on a system together with PHP at this time, but instead presume that my readers already know this. For anyone who needs assistance, additional information and support are to be found in the Appendix of this book

If these configurations have been made, Zend Framework 2 should show itself for the first time when `http://localhost` is called up in the browser:



Welcome to Zend Framework 2

Congratulations! You have successfully installed the ZF2 Skeleton Application. You are currently running Zend Framework version 2.0.0beta4. This skeleton can serve as a simple starting point for you to begin building your application on ZF2.

Follow Development

Zend Framework 2 is under active development. If you are interested in following the development of ZF2, there is a special ZF2 portal on the official Zend Framework webiste which provides links to the ZF2 wiki, dev blog, issue tracker, and much more. This is a great resource for staying up to date with the latest developments!

Discover Modules

The community is working on developing a community site to serve as a repository and gallery for ZF2 modules. The project is available on GitHub. The site is currently live and currently contains a list of some of the modules already available for ZF2.

Help & Support

If you need any help or support while developing with ZF2, you may reach us via IRC: #zftalk.2 on Freenode. We'd love to hear any questions or feedback you may have regarding the beta releases. Alternatively, you may subscribe and post questions to the mailing lists.

© 2006 - 2012 by Zend Technologies Ltd. All rights reserved.

Start page ZendSkeletonApplication of Zend Framework 2

Directory structure of a Zend Framework 2 application

Now, we can finally look at it, a `Zend\Mvc`-based Zend Framework 2 application with its characteristic directory layout and the typical configuration and initialisation code:

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

In our case the "Application Root" is the `ZendSkeletonApplication` directory, which is automatically generated by cloning the appropriate GitHub repository. In the `config` directory, there is, on the one hand, `application.config.php`, which contains the basic configuration for `Zend\Mvc` and its collaborators as a PHP array. In particular, the `ModuleManager` is configured there; we will frequently talk about its details in the course of the book. If required, the `autoload` directory contains additional configuration data in the form of additional PHP files; initially, this seems a bit strange, but one becomes accustomed to it. To begin with, the directory's designation as "autoload" is a bit irritating. In this location, "Autoload" has nothing to do with the "Autoloading" of PHP Classes, but instead indicates that the configurations that are filed in this directory will be automatically taken into consideration. And that occurs chronologically after the configuration of the `application.config.php` and also after the configurations performed by the individual modules, which we will talk about later. This sequence of configuration evaluation is extremely important because it allows the situation-dependent overwriting of configuration values. The same principle applies to `global.php` und `local.php`: configurations in the `global.php` are always valid, but they can be overwritten by configurations in the `local.php`. Technically speaking, Framework initially reads in the `global.php` and subsequently the `local.php`, whereby previously defined values can be replaced, if necessary. What is that good for? In this manner, configurations can be defined independently of the runtime environment. Let us assume that the programmers of an application have set up a runtime environment locally on their computers. Since a MySQL database is required for the application, all of the developers have installed this on their computers and in the process have configured the access rights such that passwords, which the respective developers also otherwise frequently use, are utilised. It is indeed more convenient. However, since each developer potentially has an individual password for the database, this configuration cannot be hard-wired, but must be individually specified. To achieve this, the developer enters his or her connection data in the `local.php` file, which he or she maintains locally in the computer and does not check into the code administration system either. Whereas the connection data for the "live system" are deposited in `global.php` file, every developer can work with his or her own connection data, which are defined with the aid of the `local.php`. In this manner even special configurations for test or staging systems can be deposited. Incidentally, configuration files of the form "xyz.local.php" (also applies to "global"), for example `db.local.php`, are also processed by Framework as described above.

The individual modules of the application are located in the `Module` directory. Each module comes with its own typical directory tree, which we will take a closer look at later. However, at this time the important thing is that every module can also have its own configuration. We now have three places in which something can be configured: `application.config.php`, module-specific configuration and the `global.php` and `local.php` files (or their "specialisations" as described above), which the system reads in exactly this order and ultimately provide a large, common configuration object, because in the course of execution exactly these configurations are merged. If the configurations of application.config.php are only of interest in the first few meters of bootstrapping, the configurations of the modules and those from global.php and local.php are also important in the later course of the processing chain and are generously made available by the ServiceManager. We will also learn more about this later. The attentive reader realises at this time that as a result of this "configuration cascade", for example module configurations that flow into the application from third party manufacturers' modules can be extended or even replaced. This is very practical.

The vendor directory contains conceptionally the code which one did not write oneself (ignoring the "ZendSkeletonApplication" code at this time, but which one could have had to write oneself in case of doubt) or which one did not write especially for this application. Zend Framework 2 is thus located approximately there, but, if necessary, also in other libraries. When dealing with additional libraries, one must always ensure that the corresponding classes can be addressed by the application. However, if one can install the respective library using composer, this work does not have to be done by the developer either. The installation of additional libraries should therefore in the ideal case always be performed using composer. The fact that also the ZF2 modules, which actually should be located in the module directory, can also be made available via the vendor directory is also interesting. (To be perfectly correct, one would have to say that is can be configured via application.config.php and the modules can therefore basically be deposited anywhere.) This means that third party manufacturers' libraries that adhere to the Zend Framework 2 module standard can also be added in this manner. Thus, one can ensure that only those modules that one actually developed in the scope of the respective application are located in the module directory. All other modules can also be made available via vendor

All files that are to be made externally accessible via the web server (with the exception of specific restrictions in web server configuration) are located in public. This is also the place for images CSS or JS files as well as for the "central entry point", the index.php. The idea behind this is that every HTTP request that reaches the web server and a specific application initially results in calling up the index.php. Always. Regardless of how the URL call itself is formulated. The only exceptions are URLs that refer to an actually existing file within or below the public directory. Only in this case, does the index.php not perform the execution, instead the appropriate file is read and returned. This mechanism is achieved by a typical Zend Framework .htaccess file in the public directory:

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

In order for this to function, several conditions must be fulfilled. On the one hand, the web server must be equipped with a so-called RewriteEngine[27], which must also be activated. On the other hand, the web server has to allow an application to set directives via its own .htaccess. To achieve this, the [following directive] must be exemplarily in the Apache httpd.conf'.

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

```
AllowOverride All
```

The `data` directory is relatively unspecific. Basically, data of all kinds, which have anything to do with the application (documentation, test data, etc.) or that are generated in the running time (caching data, generated files, etc.), can be deposited there.

The index.php file

Every request that does not map onto a file that actually exists in the `public` directory is thus redirected via the `.htaccess` file to the `index.php`. It therefore has a special importance for work with Zend Framework 2. At this time, it is again important to realize that the `index.php` itself is not part of Framework, but that it is indispensible for using the Framework's MVC components. Please remember: `Zend\Mvc` is the component that represents the "processing framework" for an application. The `index.php` comes with the ZendSkeletonApplication; thus, we do not have to develop it ourselves.

Because of the importance of the `index.php` — both for Framework and for our understanding of Framework's mechanics—we will now risk a detailed look at this very easily understood file:

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

Listing 4.1

To begin with, we will change to the application root directory of the application, in order to be able to easily refer to other resources. Then `init_autoloader.php` is called up; this initially triggers autoloading by composer. This nondescript call-up ensures that all the libraries that have been installed by composer automatically make their classes available via autoloading mechanisms:

```php
<?php
// [..]
if (file_exists('vendor/autoload.php')) {
        $loader = include 'vendor/autoload.php';
}
// [..]
```

Listing 4.2

Consequently, we can dispense with all `require()` call-ups in the application. The few lines that I have written down here in such an emotionless manner actually represent an enormous attainment for us as PHP developers: It simply could not be easier to integrate libraries into one's own application.

In the `init_autoloader.php`, the autoloading of the ZF2 classes via the environment variable `ZF2_-PATH` or via Git submodule is then also alternatively ensured, just in case that one did not obtain Zend Framework via composer because in that case the above-mentioned autoloading mechanism of composer is sufficient. With the aid of the environment variable `ZF2_PATH`, for example, a number of applications in the system can use a central installation of the framework code. Whether or not this is truly expedient, I cannot really say. Now, a brief check to see whether Zend Framework 2 can now be loaded—otherwise nothing will happen—and then we can get started:

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

Listing 4.3

The call-up of the class method `init()` of the `Application` initially ensures that the `ServiceManager` is superimposed. The `ServiceManager` is the central object in Zend Framework 2. It allows other objects to be accessed in many ways, is normally the "principal point of contact" in the processing chain and is also the first entry point in general. We will consider the `ServiceManager` later in greater detail. For simplicity's sake, one can initially imagine the `ServiceManager` as a sort of global directory, in which an object can be deposited under a defined key. For all those who have already worked with Framework Version 1, the `ServiceManager` thus initially presents itself as a sort of `Zend_Registry`. At this point, we should perhaps make a small leap forward. Not only previously generated object instances come into consideration as values that can be deposited in the `ServiceManager` under a stipulated key, but also "Factories", which generate the respective objects—in the context of the `ServiceManager` analogously designated as "services". The underlying idea is that these services can only then be generated when they are really needed. This procedure is termed "lazy loading", a design pattern intended to delay memory and time-consuming instancing of objects for as long as possible. Indeed, some a number of services for some types of requests are never needed; why should the always be instanced beforehand?

But back to the code: The `init()` method is transferred to the application configuration as parameter, and this has already been taken into consideration by the generation of the `ServiceManager`:

```php
<?php
// [..]
$serviceManager = new ServiceManager(
        new ServiceManagerConfig(
                $configuration['service_manager']
        )
);
// [..]
```

Listing 4.4

At this point, the `ServiceManager` is now initialised and equipped with those services which are required in the scope of processing of requests by `Zend\Mvc`. However, the `ServiceManager` can also be effectively used for completely different purposes, beyond `Zend\Mvc`.

Subsequently, the application configuration itself is deposited in the `ServiceManager` for later use.

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

Listing 4.5

Then the `ServiceManager` is asked to perform its services for the first time.

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

Listing 4.6

The `get()` method requests a service. Incidentally, in this situation we already have a case in which the `ServiceManager` does not return an instantiated object, but instead uses a "factory" to generate the requested service, by acclamation as it were. In this case, the `Zend\Mvc\Service\ModuleManagerFactory` is used, and generates the requested `ModuleManager`.

But how does the `ServiceManager` actually know now that whenever the `ModuleManager` service is requested that the above-mentioned factory is to be called upon for its generation? Let us again look at the code ahead of it:

```php
<?php
// [..]
$serviceManager = new ServiceManager(
        new ServiceManagerConfig($configuration['service_manager'])
);
// [..]
```

Listing 4.7

As a result of the transfer of `ServiceManagerConfig`, the `ServiceManager` is prepared for the use of `Zend\Mvc` and has registered exactly that factory for the `ModuleManager`, among other things. In the following chapters, we will take another look at all of this in greater detail and also look at the other services which are provided as standard.

But let us now return to the code sequence: After the `ModuleManager` has now been made available via the `ServiceManager`, the `loadModules()` method initialises all the modules activated by the `application.config.php`, If the modules are ready, the `ServiceManager` is again contacted and the "application"" service is requested from it.

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

Listing 4.8

This fact may appear a bit strange, especially since the entire processing sequence indeed originally began via a `Zend\Mvc\Application`. But it now becomes clear that its `init()` method initially only initialised the `ServiceManager`, whereas the `Application` itself is then itself generated as a service.

Now a very complex procedure, which is responsible for the processing of the request itself, begins. The "application" is prepared ("bootstrapping" occurs). Back in the `index.php`, the application is then executed and the result is returned to the calling program.

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

Listing 4.9

I have devoted a chapter in this book to the exact consideration of the request processing because of its importance, but also of its complexity. Until we get to it, we will keep this in mind: The `index.php` is the central entry point for all requests that are processed by the application. These very requests are technically rerouted to the `index.php`by `.htaccess`. The actual URL that was called up by the user is naturally maintained and is subsequently read by Framework in order to locate an appropriate controller with its action. The `ServiceManager` is at the focus of the processing and gives access to the services of the application. Therefore, we must initially generate the `ServiceManager`, before it can, in turn, give us access to the `ModuleManager`, with whose help we can bring both the registered modules and the `Application`, which is responsible for processing the requests, to life. So far, so good.

Zend Framework 2 with alternative web servers

Naturally, can alternative web server instead of Apache—such as nginx[a]—be used. In this case, only the Apache-specific configuration as well as that of `.htaccess` are to be analogously transformed, for example with the help of the "nginx rules".

[a]http://nginx.org/

Preparing one’s own module

The actual application logic, i.e. the individual pages, templates, forms, etc., are encapsulated in modules. Now that we have the executable ZendSkeletonApplication at our disposal, it is time to prepare our own first module. Because we initially have to concentrate on the individual steps that are required to prepare and encapsulate our own module, we will start with the classic module. Hello, World!

Preparing the “Hello World” module

A Zend Framework 2 module is first and foremost characterised by a defined directory structure and a few files that have to be part of every module or those that can present if needed.

```
Module.php
config/
    module.config.php
public/
    images/
    css/
    js/
src/
    Helloworld/
Controller/
IndexController.php
view/
    helloworld/
        index/
            index.phtml
```

This structure must be created in a `Helloworld` directory in the `module` directory within the application. By convention, a module is its own namespace, which thanks to PHP 5.3 we can also designate as such natively. In Framework Version 1, the pseudo-namespaces still had to be used; this resulted in very long class designations, for example in `Zend_Form_Decorator_Captcha_Word` Fortunately, with PHP 5.3 and Zend Framework 2, this problem is a thing of the past.

To begin with, we will fill the `Module.php` file with life.

```php
<?php
namespace Helloworld;

class Module
{
        public function getConfig()
        {
            return include __DIR__ . '/config/module.config.php';
        }
}
```

Listing 5.1

The `Module` class is assigned to the namespace that is stipulated by our module, in this case `Helloworld`. The class itself is a normal PHP class, which can have a series of methods, which can be called up by different Framework managers and components, for example in the scope of the initialisation. The `getConfig()` method is also among them. As already in Zend Framework 1, the "convention over configuration" approach is also extensively used in the new version. This means that there are conventions (agreements) that, if used as agreed upon, make further configuration unnecessary. In this case the following convention has been stipulated: If you implement a `getConfig()` method in your module class, it will be called up in the scope of the initialisation of the `ModuleManager`. No sooner said than done! However, our method itself does not immediately return the configuration, but to achieve this it instead reads the `module.config.php` file in the module's `config` directory, which then has the following contents.

```php
<?php
return array(
        'view_manager' => array(
            'template_path_stack' => array(
                __DIR__ . '/../view'
            )
        )
);
```

Listing 5.2

To begin with, it becomes apparent that the configuration for a module is mapped via a PHP array. There are fundamentally a large number of options as to how one can maintain configurations, for example as INI file, via YAML or as XML structure. All these structures require more or less complex parsing. However the most efficient and in Framework the preferred method is to immediately deposit the configuration in PHP code. This makes any parsing unnecessary and a slender `include()` already ensures the desired effect of reading in the configuration. But here again, "convention over configuration" also applies. If we define a `view_manager` section in our configurations, these values will always be subsequently considered when searching for the correct template as if by magic. Thus, we configure here the directory in which our module's Views (the HTML templates) will be deposited. Accordingly, at this time there is no "convention over configuration", but rather explicit information.

Moreover, we should specify in the `Module.php` how the autoloading of the individual module classes it to function. To achieve this, we implement the `getAutoloaderConfig() method that will be processed during the initialisation of the` ModuleManager'—once again, according to convention.

```php
<?php
// [..]
public function getAutoloaderConfig()
{
    return array(
        'Zend\Loader\StandardAutoloader' => array(
            'namespaces' => array(
                __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__
            )
        )
    );
}
// [..]
```

Listing 5.2

We will return to the "Autoloading" topic again later in more detail and now will have to be satisfied with the knowledge that the construct described above ensures that the classes of this module—especially also the controller—will be automatically loaded and can thus also be taken into consideration by Framework. The `Module.php` file now looks like this:

```php
<?php
namespace Helloworld;

class Module
{
        public function getAutoloaderConfig()
        {
            return array(
                'Zend\Loader\StandardAutoloader' => array(
                    'namespaces' => array(
                        __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__
                    )
                )
            );
        }

        public function getConfig()
        {
            return include __DIR__ . '/config/module.config.php';
        }
}
```

Listing 5.3

Now we will dedicate ourselves to the `IndexController` in the `/src/Helloworld/Controller` directory:

```php
<?php
namespace Helloworld\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class IndexController extends AbstractActionController
{
        public function indexAction()
        {
            return new ViewModel(array('greeting' => 'hello, world!'));
        }
}
```

Listing 5.4

To begin with, we will again pay attention to the namespace. Our `IndexController` thus belongs to the `Helloworld` module and is a `Controller` there. So far, so good. The class inherits from the Framework class `Zend\Mvc\Controller\AbstractActionController` everything that makes it what is now is: a class that can process a request ("dispatching") and in the process uses its actions. The term "actions" means public methods that again conform to a certain name convention: a method of the class then becomes an "action" when "action" is appended to the method designation. The controller now has an action, the `indexAction`.

Many things customarily now occur inside the `indexAction`, such as the processing of request parameters, writing or reading of date from databases or accessing remote web services. Normally, the action does not do everything itself (in this context one otherwise also speaks of the so-called "fat controller"), but rather delegates the individual tasks to other fellow campaigners. This approach normally increases the reusability and serviceability of the code.

As a rule, an action ends its work by making the results of the operations performed available for presentation in the calling program's browser. In Zend Framework 2 all the required data are returned (that's something that is normally only said in tennis) in the form of a so-called "view models". To put it simply, a "view model" represents the data underlying a "user interface (UI)" and additionally also controls the status of certain UI components. We will so into this in more detail again later.

When we now desire to display the salutation "hello, world!" as heading in a browser, the appropriate `h1` tag is still missing. Since the so-called "view" is responsible for the HTML presentation in a MVC-based application, we now have to create this (strictly speaking we are really only generate a "view template" that produces the desired result in the scope of "rendering" a "view" and on the basis of a "view model". For every action there is normally exactly one view or a view template, respectively. To achieve this, we enter the following in the `index.phtml` file in the `view/helloworld/index` directory:

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

Listing 5.5

Also in the structuring of the view, the module's namespace, but also the designation of the controller and that of the action, plays a role, as can be clearly seen. Our view is located in a `view` directory. That is fixed and does not change. Then in a subdirectory that is named after the module, and there again in a subdirectory that is named after the controller. The view itself bears the name of the respective action with the suffix ".phtml". A file with the ".phtml" ending comprises by convention PHP code and HTML markup, where by the PHP code is restricted to the presentation and should not, for example, contain business logic. Incidentally, the "phtml" suffix stands for PHP + HTML. In our view file, we thus have HTML markup and then access the view's data model via `$this`, which we generated in the controller beforehand. We access the key `greeting` directly with "greeting", for which we filed the value "hello, world!" in the view model. We then output it by means of `echo`.

But we are still not finished. We want to see "hello, world!" on the screen when we call-up the URL `http://localhost/sayhello`. To achieve this, we have to extend the configuration of the module in the `module.config.php` by a corresponding route and the details of our controller.

```
<?php
return array(
        'view_manager' => array(
            'template_path_stack' => array(
                __DIR__ . '/../view',
            ),
        ),
        'router' => array(
            'routes' => array(
                'sayhello' => array(
                    'type' => 'Zend\Mvc\Router\Http\Literal',
                    'options' => array(
                        'route'    => '/sayhello',
                        'defaults' => array(
                            'controller' => 'Helloworld\Controller\Index',
                            'action'     => 'index',
                        )
                    )
                )
            )
        ),
        'controllers' => array(
            'invokables' => array(
                'Helloworld\Controller\Index'
                        => 'Helloworld\Controller\IndexController'
            )
        )
);
```

Listing 5.6

Analogous to the `view_manager` key, the `router` key ensures—by convention—that the configuration of the routing of the corresponding Framework components is made accessible. Since we will consider routing later in more detail, only this much will be said at this point: At this time, we transfer an array with individual routes, one of which we have named "sayhello". It should always take effect when the "/sayhello" string follows the host information (in our case `localhost`) in the URL. If this is the case, Framework should ensure that the `IndexController`, which we have just prepared, and in it the action `index` is executed. And that is actually everything. Due to the nested array notation, the configuration initially appears to be a bit unclear. But after a short time, one has quickly become accustomed to it.

It is interesting to note that we specified `Helloworld\Controller\Index` as the value for the controller although the control is indeed named `IndexController`. The explanation is to be found somewhat further on.

```php
<?php
// [..]
'controllers' => array(
        'invokables' => array(
                'Helloworld\Controller\Index'
                        => 'Helloworld\Controller\IndexController'
        )
)
// [..]
```

Listing 5.7

With this small piece of code, we define a distinct name for our controller that is valid across all of the application's modules. To ensure that it is unambiguous, we placed the designation of the controller in front of the module's name. `Helloworld\Controller\Index` is thus now the symbolic name for our controller, which is correspondingly used in the route configuration.

Last but not least, we now have to extend the `application.config.php` file such that our new module will be considered at all. To achieve this, we amend the name of our module in the `modules` section.

```php
<?php
// [..]
'modules' => array(
        'Application',
        'Helloworld'
)
// [..]
```

Listing 5.8

The URL `http://localhost/sayhello` should now provide the desired result and output "hello, world!" to the screen.

Autoloading

As a general rule, classes must initially be made available by means of a `require()` call (or something similar) before they can be used the first time, for it is indeed so that during the execution of a script only the code that was previously made available to the PHP interpreter is accessible there. Thus, it is not enough to program a PHP class, to deposit the former somewhere in the file system and the latter at another location, to refer to the script that is being executed without having made the class known beforehand. And now a brief digression: One must absolutely differentiate between code that is part of the PHP core and the so-called "userland" code. Whereas, for example, the core class allows `\DateTime` to be used without previous registration, this does not apply for classes that you have written yourself, i.e. userland code. Such code must always be made known to the PHP interpreter initially, and the respective PHP files in which the class definitions are located must have been loaded.

Zend Framework 2 makes intensive use of autoloading. Autoloading simply means that the registration of classes is performed automatically; the respective PHP files with the classes that are defined there are thus automatically loaded when needed. In order for this to function, a certain configuration must be performed—as we have already seen in the `getAutoloaderConfig()` method in the `Module.php` file.

Indeed, we avoid a great deal of typing on every page with autoloading because we would otherwise have to load each file with an explicit `require()`; but, on the other hand, we burden the system additionally with the autoloading function. Thus, autoloading has certain costs that can make themselves felt in the execution time of a Framework 2 application. The good thing is that are different ways, among them high-performance ones, in which autoloading can be implemented, which are all supported by Framework. Framework has two essential classes that are used for autoloading.

Standard autoloader

The standard autoloader is the implementation that has meanwhile become the customary manner of realising autoloading. In this context, the class name is translated one-to-one into a file name. Consequently, the corresponding loader expects, for example, that the `Zend_Translate` class (from Zend Framework 1) is defined in a `Translate.php` file in the `Zend` directory. This also applies to classes that make use of "real namespaces": A class `Translate` that is defined in the `Zend` namespace is expected to be at the same location in the file system. This convention corresponds to both the PEAR[28]-Standard and the PSR-0[29] of the PHP Framework Interoperability Group](https://github.com/php-fig/fig-standards). The important thing is than one thinks of stating where the corresponding directory for the respective (pseudo-)namespace is located in the file system, as we did in the `Module.php` file.

```php
<?php
namespace Helloworld;

class Module
{
        public function getAutoloaderConfig()
        {
            return array(
                'Zend\Loader\StandardAutoloader' => array(
```

[28]http://pear.php.net/
[29]https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-0.md

```
                    'namespaces' => array(
                        __NAMESPACE__
                                => __DIR__ . '/src/' . __NAMESPACE__,
                    ),
                ),
            );
        }

        // [..]
}
```

Listing 5.9

The `include_path` is namely no longer consulted, which should accelerate the loading of classes to the greatest possible extent. One does no really have to know any more at this time. If one conforms to these conventions, the classes will be automatically loaded without any problems.

ClassMapAutoloader

However, the highest-performance implementation of autoloading is the `ClassMapAutoloader`; it operates on the basis of a simple, associative PHP array, which contains the fully-qualified class names as key in each case and the appropriate file names as value. It looks approximately like this:

```
<?php
return array(
        'PhlyContact\Service\ContactControllerFactory'
                => __DIR__ . '/src/PhlyContact/' .
                        'Service/ContactControllerFactory.php',
);
```

Listing 5.10

If the corresponding class is requested, the loader looks in the array for the appropriate value and loads the file. That's it. In this case the disadvantage is obvious: The class map has to be continuously maintained. If a certain class is not located there, the autoloading fails. Fortunately, there is, on the one hand, the possibility of letting a ClassMap be generated automatically (e.g. in the scope of a build process) to ensure that one did not forget any class, and, on the other hand, several methods of autoloading can be combined.

```php
<?php
public function getAutoloaderConfig()
{
    return array(
        'Zend\Loader\ClassMapAutoloader' => array(
            __DIR__ . '/autoload_classmap.php'
        ),
        'Zend\Loader\StandardAutoloader' => array(
            'namespaces' => array(
                __NAMESPACE__ => __DIR__ . '/src/' . __NAMESPACE__,
            ),
        ),
    );
}
```

Listing 5.11

In this manner, the ClassMap will be initially consulted and if necessary the PSR-0 mechanism will be resorted to if no hits have occurred up to that point.

The ClassMap itself is located in a file of its own outside the `Module.php` and is only referenced from there. The underlying idea is to generalize the procedure for the implementation of autoloading to the extent that—even outside of Zend Framework—libraries from different sources can be integrated without problems. In the last several years, some code libraries have already realized that it is a real added value for the user if the library provides some form of support for automatically loading the library's classes. The problem with this is that in each case it is again a matter of isolated solutions. In the concrete realisation of autoloading, each library goes its own way in case of doubt; and even if the ways are similar, they indeed differ in detail. In the scope of the modules of a ZF2 application, Framework orients itself on an additional standard from which, for example, the previously mentioned composer profits and in this manner to allow "translibrary" autoloading in the simplest way possible. For this purpose, every module is provided with three additional files relevant to autoloading: `autoload_classmap.php`, `autoload_function.php` und `autoload_register.php`.

The `autoload_classmap.php` returns a PHP array as mapping of class names and file names as described above, which any arbitrary autoloader—not only that of Zend Framework, but also, for example, that of composer—can process and if necessary also even combine them with ClassMaps of other libraries. In contrast, `autoload_function.php` returns a PHP function:

```php
<?php
return function ($class) {
        static $classmap = null;
        if ($classmap === null) {
            $classmap = include __DIR__ . '/autoload_classmap.php';
        }
        if (!isset($classmap[$class])) {
            return false;
        }
        return include_once $classmap[$class];
};
```

Listing 5.12

The returned function can also be processed by an autoloader, for example, in the form that it can be considered as an additional source for the autoloading of classes. Ultimately, this function also again accesses the ClassMap. And `autoload_register.php` is even more slender:

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

Listing 5.13

In this case, the previously defined autoloading function, which again accesses the ClassMap proper, is directly registered for autoloading and not returned to the calling program again. A simple

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

Listing 5.14

then insures that the autoloading for exactly these components functions. This is, however, without the option of performing further optimisations, for example in the processing sequence of all registered autoloading functions.

The three autoloading files are, as we have already seen in the previous chapter, not required for a module to function because all of these files refer to the autoloading ClassMap, which itself is not absolutely necessary, but can noticeably improve the performance of an application.

In conclusion, a complete module directory layout for the "Hello world" module complete with the autoloading files is then depicted as follows:

```
Module.php
autoload_classmap.php
autoload_function.php
autoload_register.php
config/
    module.config.php
public/
    images/
    css/
    js/
src/
    Helloworld/
Controller/
IndexController.php
views/
    Helloworld/
        Index/
            index.phtml
```

One time Request and back again

Let us look at exactly what happens to a request and how the Framework's different collaborators generate the answer with our "hello, world!" on the browser.

Everything begins with the call-up of the `http://localhost/sayhello` URL. The HTTP request reaches the web server, which after consulting `.htaccess` decides that the `index.php` has to be processed by the PHP interpreter. In the scope of the `index.php`, the autoloading is initially configured and subsequently the application's `init()` method is called up.

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

Listing 6.1

ServiceManager

The `ServiceManager` instantiated there:

```php
<?php
// [..]
$serviceManager = new ServiceManager(
        new ServiceManagerConfig($configuration['service_manager'])
);
// [..]
```

Listing 6.2

As a result of the transfer of `ServiceManagerConfig`, the `ServiceManager` is equipped with several standard services which `Zend\Mvc` requires for smooth functioning. In addition the corresponding configuration is transferred to the `ServiceManagerConfig` from the previously loaded `application.config.php`.

The `ServiceManager`is a sort of `Zend_Registry` with extended functions. Whereas the `Zend_-Registry` of Version 1 could merely file existing objects under a certain key and subsequently load them again (key value storage), the `ServiceManager` goes several steps further.

Services and service generation

In addition to the administration of services in the form of objects, a “generator” can also be registered for a key, which generates the respective service initially when needed. To achieve this, either classes (fully qualified, i.e., if necessary, with declaration of the namespace, which are then on request instantiated, for example in the form

```php
<?php
$serviceManager->setInvokableClass(
        'MyService',
        'Helloworld\Service\MyService'
);
```

Listing 6.3

or factories can be deposited.

```php
<?php
$serviceManager->setFactory(
        'MyServiceFactory',
        'Helloworld\Service\MyServiceFactory'
);
```

Listing 6.4

In order for the `ServiceManager` to be able to manage the `MyServiceFactory`, the latter has to implement the `FactoryInterface`, which requires a `createService` method. This method is then called up by the `ServiceManager`. As light-weight implementation of a factor, a Callback function can be directly transferred.

```php
<?php
$serviceManager->setFactory(
        'MyServiceFactory',
        function($serviceManager) {
                // [..]
        }
);
```

Listing 6.5

Alternatively, can an abstract Factory also be analogously filed, where by this is added without Identifier; this is shown here exemplarily in a Callback variant.

```php
<?php
$serviceManager->addAbstractFactory(
        function($serviceManager) {
                // [..]
        }
);
```

Listing 6.6

In addition, so-called "Initializers" can be filed; they ensure that a service is equipped with values or references to other objects when it is called up. "Initializers" can be so conceived that the inject objects into a service when the respective service implements a defined interface. We will see this in action again later. At the moment, we should only remember that they exist. All services made available by the `ServiceManager` are so-called "shared services", this means that the instance of a service—generated when needed or already present—can also be returned at a second request of just this service instance. The instance of the service is thus reused. This is valid for all standard services defined by Framework; the only exception in this context is the `EventManager`. If a new service is registered, one can also prevent the reuse of instances.

```php
<?php
$serviceManager->setInvokableClass(
        'myService',
        'Helloworld\Service\MyService',
        false
);
```

Listing 6.7

Standard services, Part 1

At the very beginning of request processing, the `ServiceManagerConfig` ensures that a number of standard services are made available. It is important to realize that there are services in the `ServiceManager`, which can in part only be used by Framework (or more exactly by its MVC implementation), whereas some other services are also useful for the application developer, for example in the context of a Controller. This will become clearer somewhat further along.

Invocables

The following service is made available as an "invocable", i.e. by specifying a class, which is then instantiated when necessary.

- `SharedEventManager` (`Zend\EventManager\SharedEventManager`): Allows the registration of listeners for certain events, also when the event manager required for this is not yet available. The `SharedEventManager`is automatically made available by a new `EventManager` when this is generated by the `ServiceManager`. Further explanations of the `SharedEventManager` will be found later in the book.

Factories

In the context of the `ServiceManager`, Factories are there to make services available, which do not exist until the real request occurs, but rather are built by a Factory "on demand". The following services are made available indirectly by a factory as standard.

- `SharedEventManager` (`Zend\EventManager\SharedEventManager`): The `EventManager` can generate events and inform registered listeners about them. It can also be requested via the `Zend\EventManager\EventManagerInterface`alias.
- `ModuleManager` (`Zend\Mvc\Service\ModuleManagerFactory`): Administers the modules of a ZF2 application.

Configuration

The `ServiceManager`is thus decisively controlled for use in the scope of request processing by two configurations: The `ServiceManagerConfig`, which defines a number of standard services for request processing, and also by the `application.config.php` or module-specific configurations, respectively. In each case, the `service_manager` key is essential for this:

```php
<?php
return array(
        // [..]
        'service_manager' => array(
            // [..]
            ),
        ),
        // [..]
);
```

Listing 6.8

Below the service_manager' keys, the following keys are then possible:

- `services`: Definition of Services with the aid of already instantiated objects.
- `invocables`: Definition of services by declaration of a class, which is instantiated when needed.

- `factories`: Definition of factories, which instantiate serves.
- `abstract_factories`: Definition of abstract factories.
- `aliases`: Definition of aliases.
- `shared`: Allows the explicit declaration of whether a certain service can be used a number of times or should be re-instantiated if again required.

As soon as the `ServiceManager` is available, the `application.config.php` is, then as a whole, i.e. also with the other non-`ServiceManager`relevant sections, itself made available as service.

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

Listing 6.9

This is important because other components, such as the `ModuleManager` or `ViewManager`, also access these services.

At the present time, the `ServiceManager` (equipped with diverse standard services) is thus in readiness and, in a manner of speaking, is only waiting for the show to begin. For, up to now not much has happened except for a few basic preparations. In fact, at this point nearly all of the above-mentioned services do not yet exist because they have not yet been requested and are only generated when necessary.

Writing a service of one's own

Let us draw up a service of our own for our "Hello world" module in an exemplary manner. To achieve this, we initially add another `Service` subdirectory in the `src/Helloworld` directory of our module.

```
Module.php
config/
    module.config.php
public/
    images/
    css/
    js/
src/
    Helloworld/
                Controller/
IndexController.php
```

```
        Service/
GreetingService.php
view/
    Helloworld/
        Index/
            index.phtml
```

There we create a `GreetingService` class. This class must not implement any special interfaces or be derived from any basic classes; it is thus a so-called “POPO”, a “Plain Old PHP Object”. The only important thing is that we do not forget to make the class available in the right namespace.

```php
<?php
namespace Helloworld\Service;

class GreetingService
{
        public function getGreeting()
        {
            if(date("H") <= 11)
                return "Good morning, world!";
                else if (date("H") > 11 && date("H") < 17)
                return "Hello, world!";
            else
                return "Good evening, world!";
        }
}
```

Listing 6.10

Make the service available as an invocable

To use this class as a service in our controller and to be able to display a time-oriented greeting, we must add the class to the `ServiceManager` as Service. We can do this is the scope of our module in two ways: In the course of module configuration (`module.config.php`) by adding the section

```php
<?php
// [..]
'service_manager' => array(
    'invokables' => array(
        'greetingService' => 'Helloworld\Service\GreetingService'
    )
)
// [..]
```

Listing 6.11

or programmatically by adding the `getServiceConfig()` function in the `Module.php`:

```php
<?php
public function getServiceConfig()
{
    return array(
        'invokables' => array(
            'greetingService'
                => 'Helloworld\Service\GreetingService'
        )
    );
}
```

Listing 6.12

Both ways lead to the objective. Our service is now available in the form of an "invocable". We can request the service in the `IndexController` of our "Hello World" module and use it:

```php
<?php
// [..]
public function indexAction()
{
    $greetingSrv = $this->getServiceLocator()
            ->get('greetingService');

    return new ViewModel(
            array('greeting' => $greetingSrv->getGreeting())
        );
}
```

Listing 6.13

Making the controller available via a factory class

However, we do have one problem now: The controller is dependent on a service (and the `ServiceManager`), which it actively accesses. Admittedly, it does not instantiate the class itself, which is good; thus, it does provide us with a possibility of making an alternative implementation available in the `ServiceManager`, if necessary, but it actively ensures that all dependencies have been resolved. At the latest, that will create problems for us when we desire to perform unit testing. A possible alternative in this case would be the previously mentioned “dependency injection” or “inversion of control”. In this context, the required collaborators are automatically made available and must no longer be actively requested. In the framework of the `ServiceManager`, we can realise this procedure, for example, via a preceding factory.

To achieve thus we prepare the `IndexControllerFactory` factory in the same directory in which the `IndexController` has been deposited:

```php
<?php
namespace Helloworld\Controller;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class IndexControllerFactory implements FactoryInterface
{
        public function createService(ServiceLocatorInterface $serviceLocator)
        {
            $ctr = new IndexController();

            $ctr->setGreetingService(
                    $serviceLocator->getServiceLocator()
                            ->get('greetingService')
                );

            return $ctr;
        }
}
```

Listing 6.14

In addition, we alter the `module.config.php` in the `controllers` section as follows:

```php
<?php
// [..]
'controllers' => array(
    'factories' => array(
        'Helloworld\Controller\Index'
                => 'Helloworld\Controller\IndexControllerFactory'
    )
)
// [..]
```

Listing 6.15

From now on, our `IndexController` is thus no longer generated by instantiation of a defined class, but by the deposited factory, which was previously organised by the controller's collaborators and in this case placed at the disposal of the controller by "Setter Injection". We still have to add the corresponding "Setter" to the `IndexController` and in the course of the action itself to access the corresponding member variable, instead of accessing the `ServiceLocator`.

```php
<?php

namespace Helloworld\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class IndexController extends AbstractActionController
{
        private $greetingService;

        public function indexAction()
        {
            return new ViewModel(
                    array(
                            'greeting' => $this->greetingService->getGreeting()
                        )
            );
        }

        public function setGreetingService($service)
        {
            $this->greetingService = $service;
        }
}
```

Listing 6.16

Thus, a factory class can be used for both the generation of a service and for the generation of a controller. Indeed, it is as follows: The `Zend\ServiceManager` is employed in ZF2 in several ways. Once in the form which we have already discussed: as central instance via which even the `Application` itself is generated, i.e. the "container" for the entire application, if you wish to call it that. And then there is, as we will soon discuss in more detail, a number of specialised "ServiceManagers", for example one only for the application's controller. In this context, the following lines of the `IndexControllerFactory` are of particular interest:

```php
<?php
public function createService(ServiceLocatorInterface $serviceLocator)
{
        // [..]
        $ctr->setGreetingService(
                $serviceLocator->getServiceLocator()
                        ->get('greetingService')
        );
        // [..]
}
// [..]
```

Listing 6.17

It is apparent that initially `getServiceLocator()` is invoked in the `$serviceLocator`. The reason for this is the fact that the factory's `createService()` method is always transferred to the "ServiceManager", which has been charged with the generation of the service, thus, in this case, the `ControllerLoader`(is automatically called up by Framework) which was reserved for the generation of controllers. However, this in turn does not have any access to the `GreetingService`, which we prepared beforehand and which we only made available in the "central ServiceManager" (it is indeed ultimately not a controller). In order that the services of the "central ServiceManager" can now be made available despite this, the `ControllerLoader` accesses the central ServiceManager via the `getServiceLocator()`, and makes the former available to it by just these methods. Somewhat later in this chapter you will learn more about the details of this mechanism.

Making the controller available via a factory callback

However, with the `IndexControllerFactory` we now have an additional class in our source code. For the time being, this is basically not a problem, but it could be a bit too much of a good thing in a case like this, in which it is not much of a challenging task to generate the factory. A light-weight alternative, which also makes it possible for us to avoid direct dependence, is the use of a callback function as a factory in the `module.config.php`:

```php
<?php
// [..]
'controllers' => array(
    'factories' => array(
        'Helloworld\Controller\Index' => function($serviceLocator) {
            $ctr = new Helloworld\Controller\IndexController();

            $ctr->setGreetingService(
                    $serviceLocator->getServiceLocator()
                            ->get('greetingService')
                        );

            return $ctr;
        }
    )
)
// [..]
```

Listing 6.18

The code for the generation of the `IndexController`, which was previously located in the `IndexControllerFactory`, has now been moved directly into the `module.config.php`.

Make the service available via Zend\Di

Regardless of which form of Factory is used, in all of them the generation of the respective object occurs programmatically, this means that appropriate PHP code must be written. `Zend\Di` provides an alternative option with whose help we can generate entire object graphs via configuration files. We will go into more detail later.

ModuleManager

But now let us return to request processing: After the `ServiceManager` has been adequately prepared, it has also been used for the first time, to generate the `ModuleManager` via the registered factory before loading the module is initiated.

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

Listing 6.19

Generation of the ModuleManager

The `ModuleManagerFactory` serves the purpose of providing the `ModuleManager` service. As we remember, a factory is always used when the generation of an object becomes more complex. In the scope of this generation, initially a new `EventManager` is requested from the `ServiceManager` and is placed at the disposal of the `ModuleManager`. The `ModuleManager` is therefore able to generate events and to inform registered "Listeners" beforehand. The following events are triggered by the `ModuleManager` (at a later point in time!):

- `loadModules`: Is initiated when the modules are loaded.
- `loadModule.resolve`: Is initiated for each module that is to be loaded when the necessary data are read in.
- `loadModule`: Is initiated during loading of a module for every module when the data that have been read in are exported.
- `loadModules.post`: Is initiated after all modules have been loaded.

Module-oriented listeners

However, the `ModuleManagerFactory` does much more. To begin with, it generates a large number of listeners that are registered for the above-mentioned events.

- `ModuleAutoloader`: Ensures that the `Module` class of the individual modules can be automatically loaded.
- `ModuleResolverListener`: Instantiates the `Module.php` of the respective module.
- `AutoloaderListener`: Invokes the `getAutoloaderConfig()` method in `Modules`, in order to obtain information on how the modules' classes can be automatically loaded.
- `OnBootstrapListener`: Checks to determine whether `Modules` have an `onBootstrap()` method and registers the invocation of this method for the `bootstrap` event that will be triggered at a later point in time by the `Application`.
- `InitTrigger`: Checks to determine whether `Modules` have the `init()` method at their disposal. If they do, it is invoked.
- `ConfigListener`: Checks to determine whether `Modules` have a`getConfig()` method at their disposal, which, if present, is invoked and the returned module configurations array is united with the other configurations.
- `LocatorRegistrationListener`: Insures that instances of all `Module` classes that implement the `ServiceLocatorRegisteredInterface are injected into the` ServiceManager'.
- `ServiceListener`: Calls the `getServiceConfig()`, `getControllerConfig()`, `getControllerPluginConfig()`, `getViewHelperConfig()` methods in the `Module` class, if present (or reads out die the corresponding configurations; further details on this in the following), processes the merged configurations of all modules, applies them to the m `ServiceManager` and adds further Standard-Services to the latter.

Standard services, Part 2

After a number of standard services have been made available in the course of the generation of the `ServiceManager`—among them, in addition to the`EventManager`, even the `ModuleManager` itself—the `ServiceListener` additionally registers (at the request of its factory) a colourful assortment of additional Services, which are required in the course of the request processing. At this point a brief comment is appropriate: at this point in time the operational mode of each service can and should not be completely understood! Many of the services that are registered by the `ModuleManager` at this time will cross our path again in the course of this chapter or book. The following list thus should serve much more as a reference and outlook to that which is still to come. The following services are thus—listed here according to manner of registration—registered.

Invocables

- `RouteListener` (`Zend\Mvc\RouteListener`): Listens later to the Mvc result `onRoute` and then ensures that the router is charged with the resolution on the appropriate controller.
- `DispatchListener` (`Zend\Mvc\DispatchListener`): Listens later to the Mvc result `onRoute` and then ensures that the `ControllerLoader` loads the previously selected controller and runs it.

Factories

- `Application` (`Zend\Mvc\Service\ApplicationFactory`): The `Application` (generated by the deposited factory) represents, so to speak, the entire processing chain and in general the entire application.
- `Configuration` (`Zend\Mvc\Service\ConfigFactory`): The generated `Config` service returns the merged configuration for the application. It is also available via the `Config` alias.
- `ConsoleAdapter` (`Zend\Mvc\Service\ConsoleAdapterFactory`): Service for accessing the command line.
- `DependencyInjector` (`Zend\Mvc\Service\DiFactory`): Zend Framework has its own implementation of the so-called "dependency injection", with whose help complex object graphs based on a comprehensive configuration are automatically "merged". Instead of the `DependencyInjector` keys, one can also use its aliases `Di` or `Zend\Di\LocatorInterface`. We will look at `Zend\Di` in more detail later.
- `Router`, `HttpRouter`, `ConsoleRouter` (`Zend\Mvc\Service\RouterFactory`): Based on the request URL, the factory-generated `Router` service determines the controller that is to be invoked—if necessary, also in the "command line mode".
- `Request` (`Zend\Mvc\PhpEnvironment\Request`): Provides access to all request information, e.g. the request parameters.
- `Response` (`Zend\Http\PhpEnvironment\Response`): Represents the answer generated in the course of processing to the client.

- `ViewManager`: The `ViewManager` performs a function for the administration of views and their processing that is similar to that performed by the `ModuleManager` for the modules and the `ServiceManager` for the services. It ensures that the data will sometime become, for example, web pages with HTML markup.
- `ViewJsonRenderer` (`Zend\Mvc\Service\ViewJsonRendererFactory`): Allows the realisation of RESTful[30] controllers and thus of web services, which conforms to the REST architecture style. This topic is discussed in a chapter of its own in the book.
- `ViewJsonStrategy` (`Zend\Mvc\Service\ViewJsonStrategyFactory`): Ensures that the `ViewJsonRenderer` will be invoked when required. In the scope of this `Strategy`, for example, the system checks to see whether the `ViewModel` returned by the controller is of the `JsonModel` type.
- `ViewFeedRenderer` (`Zend\Mvc\Service\ViewFeedRendererFactory`): Allows the realisation of RSS or Atom feeds of the "view data" returned by a controller.
- `ViewFeedStrategy` (`Zend\Mvc\Service\ViewJsonStrategyFactory`): Ensures that the `ViewFeedRenderer` will be invoked when required. Part of this `Strategy` the determination of whether the `ViewModel` returned by the controller is of the `FeedModel` type.
- `ViewResolver` (`Zend\Mvc\Service\ViewResolverFactory`): Makes it possible to find "view templates".
- `ViewTemplateMapResolver` (`Zend\Mvc\Service\ViewResolverFactory`): Makes it possible for the `ViewResolver` to find View-Templates on the basis of a map.
- `ViewTemplatePathStack` (`Zend\Mvc\Service\ViewTemplatePathStackFactory`): Makes it possible for the `ViewResolver` to find View-Templates on the basis of a list of paths.

And additionally:

- `ControllerLoader` (`Zend\Mvc\Service\ControllerLoaderFactory`): The`ControllerLoader` can load a controller that was previously localised by a routing.
- `ControllerPluginManager` (`Zend\Mvc\Service\ControllerPluginManagerFactory`): Makes the `ControllerPluginManager` and, thus, a number of plugins, which can be used in controllers, are available; among them, for example, the `redirect` plugin by means of which forwarding can be realised. This service can also be requested via the `ControllerPluginBroker` keys, `Zend\Mvc\Controller\PluginBroker` or `Zend\Mvc\Controller\PluginManager`.
- `ViewHelperManager` (`Zend\Mvc\Service\ViewHelperManagerFactory`): Generates the `ViewHelperManager`, which is responsible for the administration of so-called "view helpers".

The latter three services are particularly interesting, because they, in turn, comprise the new "ServiceManager", termed "Scoped ServiceManager" in ZF jargon. Whew, now things are beginning to get a bit complicated! So let's take a slow look at things—step by step. To begin with we should remember that there is the one "central ServiceManager" in the system. All of the important "application services" are generated by using it. It is both a `ServiceManager` in a technical sense

[30] http://de.wikipedia.org/wiki/Representational_State_Transfer

and the conceptional "central ServiceManager" for us. However, there are specific services that the `ServiceManager` itself does not provide, but instead are made available by specialised "sub-ServiceManagers" or "scoped ServiceManagers", respectively, which can also provide services via the known mechanisms, i.e. "invocables", "factories", etc. All of them are also `ServiceManagers` in a technical sense.

In this context, let's again take a look at the last chapter, in which we wrote our own controller. There we find the following passage in the `module.config.php`:

```php
<?php
// [..]
'controllers' => array(
        'invokables' => array(
                'Helloworld\Controller\Index'
                        => 'Helloworld\Controller\IndexController'
        )
)
// [..]
```

Listing 6.20

When this configuration fragment is interpreted, this results in reference to the appropriate controller class under the `Helloworld\Controller\Index` key, which is registered as an "invocable" in the `ControllerLoader`, one of the standard "scoped ServiceManagers". Thus, if this controller is subsequently identified as appropriate in the scope of routing and must then be instantiated, the system uses the `ControllerLoader` to do this. In this context, one can then also characterise a controller as a service.

This procedure of the specialised Sub-ServiceManager has several advantages for certain types of services. For example, in this manner the central ServiceManager for the application services is itself not overloaded with innumerable services, and it is easy to determine all of the controllers, a task that would otherwise not be nearly as easy. Here are the different ServiceManagers again at a glance:

- Application Services (`Zend\ServiceManager\ServiceManager`): Configuration via the `service_-manager` key or the `getServiceConfig()` method (defined in the `ServiceProviderInterface`).
- Controllers (`Zend\Mvc\Controller\ControllerManager`): Configuration via `controllers` key or `getControllerConfig()` method (defined in `ControllerProviderInterface`). It can be obtained in the "central ServiceManager" via the `ControllerLoader` service designation.
- Controller plugins (`Zend\Mvc\Controller\PluginManager`): Configuration via the `controller_-plugins` key or `getControllerPluginConfig()` method (defined in `ControllerPluginProviderInterface`). It can be obtained in the "central ServiceManager" via the `ControllerPlugin manager` service designation.

- "View helpers" (`Zend\View\HelperPluginManager`): configuration via the `view_helpers` key or the `getViewHelperConfig()` method. (defined in the `ViewHelperProviderInterface`). It can be obtained in the "central ServiceManager" via the `ViewHelperManager` service designation.

Loading the modules

After the `ModuleManager` has been prepared and the required listeners have been registered, the actual loading of the modules is initiated by invocation of the `loadModules()`method of the `ModuleManager`. At this time, relatively little really happens here because the actual processing, for example the invocation of the above-mentioned methods of the `Module.php`, indeed occurs in the many registered listeners. Initially, the `loadmodules.pre` event is triggered, and then the `loadModule.resolve` event and `loadModule`, for every activated module. Finally, the `loadModules.post`event is again triggered. And that was really everything.

The Module Event Object

The concept of the `EventManager` is that in addition to being the trigger for an event and the listeners registered for the event (receivers), the event itself—represented as independent object—still exists. It is made available to all listeners. This object serves to transfer additional event-relevant information, for example a reference to the location in the code where the event is triggered. Moreover, additional data, which are helpful for the event processing in the listeners, can be transferred Thus, the module which is now being loaded is generally of interest for a listener during `loadModule`.

To do justice to the fact that, depending on the context of the event, other data are of interest, the `EventManager` of Zend Framework 2 permits deposition of situation-dependent special event classes. For the event principle in the scope of the `ModulManager`, there is as special `ModuleEvent` class, which for example bears both the module in question and additionally the name of the module.

Activation of a module

In order for a module to be taken into account at all, an explicit activation in `application.config.php` in the `config` directory is required:

```php
<?php
return array(
        'modules' => array(
                'Application',
                'Helloworld'
        )
);
```

Listing 6.21

Methods of the module class

As we have seen, a large number of methods in the `Module` class of a module are invoked if we have implemented them. In Framework there are two relevant possibilities of finding out whether this is the case. Either the `Module` class implements a specific interface (this can indeed be tested via `instance of) or the respective method is simply implemented (this can be checked via the` method_exists()‘ invocation).

Let’s now again take a detailed look at the methods in the `Module` class, which are automatically invoked by Framework and can be used by application developers:

- `getAutoloadingConfig()` (defined in `AutoloaderProviderInterface`): We have already created this method in our Helloworld module. It provides information on how the classes of the module can be automatically loaded. If we omit this method, the classes of the module (for example its controller) normally cannot be loaded and serious problems occur when the corresponding URL is invoked. Consequently, it should always be ensured that information on how the classes can be automatically loaded has been made available to Framework. Incidentally, in a purely technical context, Framework takes the information to incorporate an appropriate loader implementation for this module via `spl_autoload_register()`.
- `init()` (defined in `InitProviderInterface`): This method allows the application developer to initialise his or her own module, thus, for example, to register his or her own listeners for certain events. If necessary, the `ModuleManager` is consigned to the method and the latter can thus access the appropriate events (of the `ModuleManager`) or access the modules. The important thing is that this “method” is always invoked, that means for every request—and indeed for every module. One should also realize that this is a good place to ruin the loading time of an application. Thus, only very few and ideally only light-weight operations should be performed in the scope of the `init()` method. If one is attempting to improve the speed of a ZF2 application, one should always first take a look at the `init()` methods of the activated modules.

Here is an example for the use of the `init()`method:

```php
<?php
namespace Helloworld;

use Zend\ModuleManager\ModuleManager;
use Zend\ModuleManager\ModuleEvent;

class Module
{
        public function init(ModuleManager $moduleManager)
        {
                $moduleManager->getEventManager()
                        ->attach(
                                ModuleEvent::EVENT_LOAD_MODULES_POST,
                                array($this, 'onModulesPost')
                        );
        }

        public function onModulesPost()
        {
                die("Modules loaded!");
        }

// [..]
}
```

Listing 6.21

- `onBoostrap()` (defined in `BootstrapListenerInterface`): An additional option for the application developer to implement module-specific bootstrapping. Fundamentally, this method has the same purpose and utility as `init()`, but the `onBootstrap` is invoked later in the processing; namely, when the `ModuleManager` has already finished its work and has turned the rudder over to the `Application`. Thus, when using the `onBootstrap()`, method, services and data which were not yet accessible in the `init()` are available.
- `getConfig()` (defined in `ConfigProviderInterface`): We are also already familiar with this method. It provides the possibility of referring to the module-specific configuration file, which according to convention is termed `module.config.php` and is deposited in this module in the `config` subdirectory. However, this is not obligatory. Strictly speaking, this method is absolutely required in order for a module to be executable, but, in practice, one cannot get along without a module-specific configuration file, which one makes accessible to Framework via this module. With regard to configuration, Framework allows a certain amount of flexibility. Thus, either all configurations can be made available in one or more external files via `getConfig()` or special "Config methods" can be implemented in the `Module` class. The

latter refer to the "ServiceManagers" that are present in the system, i.e. to the `ServiceManager`, `ControllerLoader`, `ViewHelperManager` and `ControllerPluginManager`.

- `getServiceConfig()`: Allows the configuration of the `ServiceManager` and is equivalent to the "config array key" `service_manager` in `module.config.php`.
- `getControllerConfig()`: Allows the configuration of the `ControllerLoader` and is equivalent to the "config array key" `controllers` in `module.config.php`.
- `getControllerPluginConfig()`: Allows the configuration of the `ControllerPluginManager` and is equivalent to the "config array key" `controller_plugins` in `module.config.php`.
- `getViewHelperConfig()`: Allows the configuration of the `ViewHelperManager` and is equivalent to the "config array key" `view_helpers` in `module.config.php`.

In this context another example: our own ViewHelper (more on the concept of the "view helper" on the following pages) can either be made known in the scope of the `module.config.php` as follows

```php
<?php
'view_helpers' => array(
    'invokables' => array(
        'displayCurrentDate'
                        => 'Helloworld\View\Helper\DisplayCurrentDate'
    )
)
```

Listing 6.22

or in the `Module.php` with the aid of the appropriate method:

```php
<?php
public function getViewHelperConfig()
{
    return array(
        'invokables' => array(
            'displayCurrentDate'
                            => 'Helloworld\View\Helper\DisplayCurrentDate'
        )
    );
}
```

Listing 6.23

Application

Now, where the `ServiceManager` has been equipped with the required services, and the `ModuleManager` has loaded the application's modules, the `Application` itself can be started and the request processing, initiated. This occurs in 3 steps, partly in the `init()` method of the application itself and partly in the `index.php`: Starting the application (`bootstrap()`), followed by the execution (`run()`) and last but not least returning the generated results (`send()`):

```php
<?php
$application = $serviceManager->get('Application');
$application->bootstrap()
$application->run();
$application->send();
```

Listing 6.24

Generation of the application & bootstrapping

The application object is generated via the factory that is registered in the `ServiceManager`. In the scope of the generation, the `Application` invokes a number of standard services, among them both the `Request`, as the basis for further processing, and the `Response`, which is to be filled with life in the scope of the processing. In addition, the `Application` gets a reference to the `ModuleManager` and its own instance of the `EventManager` (which is indeed deposited in the `ServiceManager` such that it is not shared; thus a new instance of this service is returned). The latter ensures—analogously to the `ModuleManager`—that the `Application` can trigger events and inform listeners. The `Application` sets off a number of events in the scope of the processing.

- `bootstrap`: Is executed when the application is started.
- `route`: Occurs when a controller and an action are determined for the URL.
- `dispatch`: Takes place when a determined controller is identified and invoked.
- `render`: Occurs when the result for the return is prepared on the basis of templates.
- `finish`: Is executed when the application has been completed.

In the course of bootstrapping the `Application` likewise registers (incidentally in this case the `Application` itself and not the `ApplicationFactory`) a number of listeners, which it also obtains via the `ServiceManager`: `RouteListener` for the `route` event, `DispatchListener` for the `dispatch` event and the `ViewManager` for a colourful bunch of additional listeners, which perform the processing of templates and layouts. More about this in the next section.

Furthermore, the `Application` then creates the MVC-specific event object (`MvcEvent`), which is registered with the `EventManager` as event object. `MvcEvent` then enables the listeners to access `Request`, `Response`, `Application` and the `Router`.

Finally, the `bootstrap` event is initiated and the registered listeners are run. They can also particularly be the application's individual modules, which have registered for just this event.

Execution

The `run()` method, which is executed subsequent to bootstrapping, then actuates a number of "levers" that had already been placed in the correct position. To begin with, the `Application` triggers the `route` event. The `RouteListener`, which was registered for this event beforehand, is run and the `Router` from the `MvcEvent` is asked to perform its services: i.e. to match the URL to a defined route. In this case, a route is the description of a URL on the basis of a defined pattern. We will take a detailed look at the mechanics of routing later. At the moment, we only have to remember that the `Router` now either finds a route that fits the invoked URL and thus determines the appropriate controller as well as the appropriate action or, on the contrary, the `Router` returns with bad news and did not turn up any search results at all. But let's initially remain on the successful path in this case. The appropriate route is deposited in the form of a `RouteMatch` object in `MvcEvent` by the `RouteListener`. `MvcEvent` is thus increasingly proving to be a central object in which a number of other important objects and data are available. Then the `dispatch` event is initiated, the `ControllerLoader` is invoked and uses the `MvcEvent` to find the identified controller, which is to be instantiated. To achieve this, the `DispatchListener` requests the `ControllerLoader` from the `ServiceManager` (which, technically speaking, in its own right is also again a "ServiceManager") and then the actual controller from it (i.e. from the `ControllerLoader`). In the course of this, the controller is also equipped with an `EventManager` of its own. Now, it can thus actuate events and manage listeners. Then the controller's `dispatch()` method is invoked, and performs the further processing itself. If any intermittent problems occur in this enterprise, the `dispatch.error` event is triggered; otherwise, the result of the `dispatch()` invocation is deposited in `MvcEvent` and additionally returned and thereupon the dispatch process within the controller is concluded.

In Zend Framework, controllers are so conceived that they only have to have one`dispatch()` method at their disposal. This is externally invoked, in the process the `Request`object is transferred, and the controller is expected to return an object of the `Zend\Stdlib\ResponseInterface` type when the work has been completed. In order for the principle of controller and action to function, as one is accustomed to and expects, this logic must be implemented in the controller itself; otherwise only the `dispatch` method would be invoked, but not the appropriate "action" method. To insure that that one does not have to do this oneself, one's own controller inherits this from the `Zend\Mvc\Controller\AbstractActionController`. Subsequently, any arbitrary actions can be deposited in the controller when one adheres to the convention that the method name must end with "action":

```php
<?php

namespace Helloworld\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class IndexController extends AbstractActionController
{
        private $greetingService;

        public function indexAction()
        {
                return new ViewModel(
                        array(
                                'greeting' => $this->greetingService->getGreeting(),
                                'date' => $this->currentDate()
                        )
                );
        }
}
```

Listing 6.25

Back in the `Application` , the two results `render` and `finish` are now initiated and the `run()` method concluded. Incidentally, the following fact is very interesting and helpful.: Normally, an action returns an object of the `ViewModel` type at the end of processing. It thus implicitly signals the subsequent processing steps that the result must still be processed before it can be returned.

```php
<?php
public function indexAction()
{
    return new ViewModel(
        array(
            'greeting' => $this->greetingService->getGreeting(),
            'date' => $this->currentDate()
        )
    );
}
```

Listing 6.26

However, a very practical implementation detail is the fact that when a `Response` object is returned instead of a `ViewModel`, the downstream `render` activities are omitted.

```php
<?php
public function indexAction()
{
    $resp = new \Zend\Http\PhpEnvironment\Response;
    $resp->setStatusCode(503);
    return $resp;
}
```

Listing 6.27

This mechanism is helpful if one desires, for example, to briefly return a 503 code, because the application is just undergoing scheduled maintenance or when one desires to return data of a specific Mime type, for example the contents of an image, of a PDF document or something similar.

ViewManager

Before the processing has ended, the ViewManager comes into play again. In the previous section, we have already seen that the `onBootstrap()` method is executed in the scope of the bootstrapping of the `Application` (because it is registered for the corresponding event) and that an entire series of additional preparations are made there. Up to now, we have blended this out for simplicity's sake, but now we also have to look at the details in this case. After the `ViewManager` with its many collaborators has completed its work, we have actually worked our way through the entire processing chain once.

To begin with, the `ViewManager` obtains the `Config` from the `ServiceManager`, which at this time already represents the merged "total configuration" of the application and of the modules. The `ViewManager` looks for the `view_manager` key and uses the configurations deposited there. Then the old game of registering diverse listeners and the provision of additional services begins again.

View-oriented listeners

The following listeners are generated and all of them are attached to the `dispatch` event of the `ActionController` class (or more exactly: of the `EventManager` of the `ActionController`):

- `CreateViewModelListener`: Ensures that, after execution of the controller, an object of the `ViewModel` type is available for the rendering, even if only `NULL` or an array was made available by the controller.
- `RouteNotFoundStrategy`: Generates a `ViewModel` for the case that no controller was determined and no "View Model" could be generated (404 error).
- `InjectTemplateListener`: Adds the appropriate template to the ViewModel for subsequent rendering.

- `InjectViewModelListener`: Adds the `ViewModel` to `MvcEvent`. This listener is also registered for the `dispatch.error` event of the `Application` in order to also be able to make a `ViewModel` available in case of error.

The `dispatch` event is incidentally somewhat nasty: it occurs twice in the system It is once triggered by the `Application`, and again by the `ActionController`. Even if the designation of the event is identical (it can indeed be freely selected), due to the fact that it is triggered by different EventManagers, we are dealing with two completely different events.

Incidentally, at this time, the `EventManager` of the respective , specific manifestation of the `ActionController` does not yet exist because the latter has not yet been generated at all. In this case, this problem is avoided by using the `SharedEventManager`. With the aid of the `SharedEventManager`, listeners for the events of an EventManager, which does not even exist at the time of registration, can be registered. For the time being, we'll simple leave things as they are. We'll take a more detailed look at how this mechanism is realized in the next chapter.

View-oriented services

In addition, a number of view-oriented services are made available in the `ServiceManager`:

- `View` and `View Model`: To begin with there is the `View` itself with its `View Model`, the representation of the "payload" generated from a request for the response.
- `DefaultRenderingStrategy`: Can access the `View` and is registered for the `render` Event of the `Application`. If this event occurs, the `View` is transferred to the `ViewModel`, which is obtained from the `MvcEvent` and then prompts the `View` to render just that.
- `ViewPhpRendererStrategy`: However, the actual rendering is not performed by the `View`itself, but is instead delegated to the ViewPhpRendererStrategy`, which initially specifies the appropriate renderer and transfers the finished result to the `Response‘ subsequent to processing.
- `RouteNotFoundStrategy`: Defines the appropriate behaviour in case of a `404` situation.
- `ExceptionStrategy`: Defines the appropriate behaviour in case of a "dispatch errors".
- `ViewRenderer`: Takes over the actual rendering work, i.e. the merging of the `ViewModel` data and the appropriate template.
- `ViewResolver`: In order to localise the respective template, the `ViewRenderer` accesses the `ViewResolver`.
- `ViewTemplatePathStack`: Makes it possible for the "resolver" to localise a template on the basis of deposited paths.
- `ViewTemplateMapResolver`: Makes it possible for the "resolver" to localise a template on the basis of a "key value assignment".
- `ViewHelperManager`: Makes it possible to access "ViewHelpers" in templates; this simplifies the generation of dynamic markup.

In case of an error in the scope of routing or in the course of dispatching, respectively, the `Application` triggers a `dispatch.error` event. This signalises that an error has occurred in the processing.

Summary

At first glance, the relationships appear very complex; the implementation of the MVC pattern in Zend Framework initially feels somehow over-engineered. The main reason for this feeling is the fact that, on the one hand, the entire processing procedure is broken down into extremely small individual steps, which are represented via individual classes in each case, and which must also be chronologically and contentually "orchestrated" in order that they also ultimately meaningfully interact—to the extent that is required. On the other hand, the excessive use of events and listeners makes it difficult to understand the processes and relationships within the application. Nor does the use of numerous design patterns t exactly contribute to comprehension, particularly in the beginning, especially not when one is not yet accustomed to them.

Isn't it possible to simplify things greatly? Must MVC implementation really always be so complex? Th quick, unreflected answer to these questions would be: Yes. No. There is a large number of so-called "Micro MVC Frameworks", such as Silex[31] or MicroMVC[32], which at first glance appear to be able to achieve similar results with significantly less complexity than the MVC implementation in Zend Framework 2. However, this is only true at first glance.

To begin with, `Zend\Mvc` is actually much more than "MVC". It is in reality an application platform that allows 1) the simple and effective integration of additional function in the form of one's own or third party modules, 2) the modification or complete restructuring of request processing in nearly any arbitrary manner, and 3) which, as a result of its loose coupling approach to individual components and services, also allows fulfilment of the requirements of company applications with regard to maintainability and extensibility as well as testability. `Zend\Mvc` achieves an environment of software components and is able elevate the abstraction level, on which an earlier application developer moved about in the development of web applications with PHP, to a new, substantially more productive one. At least, it is beginning to achieve just that. Whether this will really succeed must be proven. Is Zend Framework 2 the right software for my project? I think that this question is more difficult to answer for the Version 2 than it was for Version 1. The advantages of Version 2 are particularly aimed at the profession application, which does not always exist. Is Zend Framework 2 the right software for my company application in the web? Yes, I would categorically say that in any case.

Let's summarise this chapter and the course of request processing again briefly. To begin with, the request lands at the `index.php` via the use of URL rewriting (for example, via "mod_rewrite" and the appropriate `.htaccess` file). The autoloading is configured there, which thus ensures that both Zend Framework itself, but also, if need be, additionally used libraries function properly.

[31]http://silex.sensiolabs.org/
[32]http://micromvc.com/

Subsequently, the `Application` is started, which initially ensures that the `ServiceManager`, the central “service access”, is generated and equipped with important services: the `ModuleManager` and the `EventManager`. Furthermore, the `SharedEventManager`is made available As in the case of the `ModuleManager` and the `EventManager`, made available frequently means that to begin with only the factories, which are consulted for the subsequent generation of actual services in each case, are made known. Why should we detour via factories? On the one hand, because they allow us to exchange the specific implementation of the respective service if necessary. And on the other hand, because in the scope of service generation, not only the service itself is generated, but also a number of listeners, which are registered for the subsequent processing events, as is the case for the `ModuleManager`. Indeed, the `ModuleManagerFactory`, `ApplicationFactory` and the `ViewManager` perform in a similar manner in the generation of services. They initially generate the actual service, then a number of additional (sub-)services, which the service will subsequently access, and finally one or more listeners for the events of one’s own or other services. The listeners then take over a specific task themselves or refer back to the services.

This procedure ensures that the methods `run`, `loadModules`, `bootstrap`, & co. of the `ModuleManager`, `Application` & co. are very lean and really don’t do anything themselves other than to trigger the events. Everything else then passes via the listeners to the services. During its execution, the `ModuleManager` initially triggers the following events: `loadModules.pre`, the `loadModule.resolve`, the `loadModule` and the `loadModules.post`. Subsequently, the `Application` sets off the `bootstrap`, `route` and `dispatch`events; subsequently the controller with its own `dispatch` event (not to be confused with that of the `Application`); finally the `Application` again with `render`, before the `View` reports with `renderer` and `response`. Last but not least, the `Application` ends the firework event with `finish`. In case of an error in the scope of routing or in the course of dispatching, respectively, the `Application` triggers a `dispatch.error` event. This signalises that a problem has occurred in process and that the appropriate error treatment has been activated.

EventManager

We saw in the previous chapter the extent to which the Framework as a whole is based on the idea of event triggers, event objects and event listeners. In the process, the appropriate objects or "managers", each makes its own `EventManager` available, which manages the events of the respective object and also allows the addition and subtraction of listeners. Because the `Zend\EventManager` plays such an important role for the function of Framework, but also because it can be very useful in the development of one's own application, we will take a detailed look at it in the following.

Registering a listener

In particular, the EventManager provides two methods which are interesting for listeners:

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

Listing 7.1

With the `attach()` method, a listener can be registered for a specific event, whereas `detach()` removes a listener. The listeners that are registered for an event are informed in sequence. Whereby, in this context, "informed" means that the respective registered Callbacks—which, as one is also accustomed from the native PHP functions, in addition to functions, class methods, and object methods may also be closures—are invoked.

In this context, the designation of the event for which the listener should be registered must be initially specified. As convention in the designation of an event, one frequently resorts to the magic constant `__FUNCTION__` so that the method that is triggered in the event also becomes the name giver for the event itself. However, this is not required: the name can be freely selected:

```php
<?php
$greetingService->getEventManager()->attach(
        'event1',
        function($e) {
                // [..]
        }
);
```

Listing 7.2

In this case, a callback function is registered for the "event1" event. Instead of a string, an array with several event designations can also be transferred in the course of the registration if one desires to register one listener for a number of events.

```php
<?php
$greetingService->getEventManager()->attach(
	array('event1', 'event2'),
	function($e) {
		// [..]
	}
);
```

Listing 7.3

Registering several listeners at the same time

However, it is not only possible to register one listener for a number of events in one go, but also to register a number of listeners for one or even concurrently for several events in one fell swoop. To achieve this one uses so-called "listener aggregates". They are particularly helpful when one desires to group the registration of individual listeners logically. To do this, one makes a class of one's own available, which implements the `Zend\EventManager\ListenerAggregateInterface` and thus has an `attach()` and a `detach()` method at its disposal. The individual listeners are then registered there en bloc:

```php
<?php
namespace Helloworld\Event;

use Zend\EventManager\ListenerAggregateInterface;
use Zend\EventManager\EventManagerInterface;

class MyGetGreetingEventListenerAggregate
	implements ListenerAggregateInterface
{
	public function attach(EventManagerInterface $eventManager)
	{
		$eventManager->attach(
			'getGreeting',
			function($e){
				// [..]
			}
		);

		$eventManager->attach(
			'refreshGreeting',
			function($e){
```

```
                                        //[..]
                              }
                    );
          }

          public function detach(EventManagerInterface $events)
          {
                    // [..]
          }
}
```

Listing 7.4

Adding the listeners then becomes a one-liner, because the `attach()` method does all the heavy lifting and is called automatically:

```
<?php
$greetingService
          ->getEventManager()
          ->attach(
                    new \Helloworld\Event\MyGetGreetingEventListenerAggregate()
          );
```

Listing 7.5

Removing a registered listener

An already registered listener can be removed by means of the `detach()` method of the `EventManager`. To achieve this, one can use a `ListenerAggregateInterface` for `detach()` in a manner analogous to `attach()` or remove the listeners individually. To do this, one consigns the respective `CallbackHandler`, which were, for example, returned as the result of `attach()`, to `detach()`:

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

$greetingService->getEventManager()->detach($handler);
```

Listing 7.6

Trigger an event

The following invocation is sufficient to trigger an event if an `EventManager` is available in the `eventManager` member variable of the object.

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

Listing 7.7

All the listeners that have been registered for this event are now invoked. The entire process naturally occurs sequentially and blockingly. Each listener is individually invoked and only subsequent to complete processing is the next listener processed. In the process, the processing sequence is stipulated by the listener's priority, which can be declared for a listener at `attach()` or, however, by the sequence in which the listeners were added.

The `trigger()` method can be invoked, as shown above, with a string that represents the name of the respective event, or, however, when a corresponding event object is consigned.

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

Listing 7.7

Incidentally, the invocation of `trigger()` returns a result of the `ResponseCollection` type; namely everything that the invoked listeners have previously returned individually is returned collectively. By using its `first()` and `last()` methods, it is possible to access the most important return values, and to check for a specific return value in the collection with `contains($value)`. In this context, the execution sequence of the listeners results—as already described above—either from priority explicitly specified in the scope of `attach()` or alternatively simply from the sequence with which the listeners were added. In this context, the FIFO principle applies: first in, first out. The listener that was added first will thus be executed first.

And another interesting thing: the processing of the individual listeners can also be interrupted intermediately. But what is it good for? Let's look at an example: imagine that you operate a website on which users can write a reviews of a books. In order to be able to assign them properly, you initially retrieve the ISBN of the book and thus ensure that the book's title, author, etc. do not have to be manually input by the user. To do this you either access data that are already in your database (however, that is only then the case when at least one review of the book has already been written) or you obtain the book's details from a remote web service, for example from Amazon. How can you now load your data in the most efficient manner? You would prepare two listeners, which you would register for the onBookDataLoad event of your application, whereby the first listener looks for the desired data in your database and the second, in a remote web service. However, the second listener is only then executed when the first one was not successful. To achieve this, the execution of the event can be extended by a callback, which will be checked for a specific return value:

```php
<?php
$results = $this->events()->trigger(
        __FUNCTION__,
        $this,
        array(),
        function ($returnValue) {
                return ($returnValue instanceof MyModule\Model\Book);
        }
);
```

Listing 7.8

If the first listener returns an object of the MyModule\Model\Book type, the book's details have been successfully loaded, and the processing ends at this time. Alternatively, the processing can also be ended within a listener if the stopPropagation() method of the event object is actively invoked.

SharedEventManager

The SharedEventManager is the solution to the following problem: What does a listener do if it desires to register itself with an event trigger that does not yet exist at the time of the desired registration? In practice, this problem arises when a module wants to register a callback for the controller event "dispatch" in the scope its init() or onBootstrap() method. At this time, the controller with its EventManager manager has not yet been brought to life. It is thus simply impossible.

If a new EventManager is requested via the ServiceManager, the latter ensures that the former is additionally given a reference on the divided SharedEventManager for all the EventManager instances that have been generated in this manner. Listeners for specific events can be registered in the SharedEventManager under declaration of a listener "identifier". In this context, it initially does not make any difference whether the EventManager, which will set off the event later, already exists or not. The only important thing is the fact that one uses an "identifier" when registering a listener and that the respective EventManager later recognizes and executes it. Let's look at a definite example which will clarify the principle. When the init() method of the Module class of our Helloworld module is executed, the Application does not yet exist. It will be first given the breath of life when the module has been completely loaded. If we thus now desire to register a listener for the route event of the Application, we have to do this via the SharedEventManager, we have no other choice:

```php
<?php
// [..]
public function init(ModuleManager $moduleManager)
{
    $moduleManager
                ->getEventManager()
                ->attach(
                        ModuleEvent::EVENT_LOAD_MODULES_POST,
                        array($this, 'onModulesPost')
                );

    $sharedEvents = $moduleManager
                ->getEventManager()->getSharedManager();

    $sharedEvents->attach(
                'application',
                'route',
                function($e) {
                        die("Event '{$e->getName()}' wurde ausgeloest!");
                }
    );
}
```

Listing 7.9

We are now in the `Module.php` file of the `Helloworld` module. Via the den `ModuleManager`, we reach its `EventManager` and via the latter we in turn, the `SharedEventManager`, which is automatically shared by all the "EventManagers" that have been generated by the `ServiceManager`. There we now register a callback (exemplarily in the form of a closure) for the `route` event that will trigger/start the `Application` or its `EventManager`, respectively, at sometime in the future . The `application` key in this case is a convention, a string, that one must know. At the moment, in which the `Application` subsequently triggers the `route` event with its own `EventManager`—which we could not use for the registration of the listener up to this time—all of the listeners that have registered themselves by the `SharedEventManager` for the `application` key and the corresponding event will also be informed. That's extremely practical!

The following "identifiers" are preconfigured for the individual Framework components:

- For the `ModuleManager`: `module_manager`, `Zend\ModuleManager\ModuleManager`.
- For the `Application`: `application`, `Zend\Mvc\Application`.
- `AbstractActionController` (the basic class for one's own "controller"): `Zend\Stdlib\DispatchableInterfac` `Zend\Mvc\Controller\AbstractActionController`, the first part of the controller's namespace (for `Helloworld\Controller\IndexController` that would be, for example, `Helloworld`).

- View: Zend\View\View

The SharedEventManager is thus particularly appropriate for situations in which the EventManager, which would actually be responsible and which one would desire to use for the registration of a listener, is not yet available.

Besides the above-mentioned "identifiers", which are automatically generated by Framework, additional identifiers can be defined for one's own purposes.

Using events in one's own classes

The Zend\EventManager allows a class to become a trigger for events and to administer listeners. Its function is not restricted to classes and other Framework components—quite the contrary: Zend\EventManager is highly appropriate to even serve as an independent implementation of event-controlled processing.

To do this, one's own class, which should trigger events, must merely maintain an instance of the Zend\EventManager in a member variable. Let's again use the GreetingService that we used above. Assuming that we would like to write an entry in our logfile whenever a date has been generated such that we always know how frequently this service was invoked in a specific time interval. How could we realize this? Let's take a look at one possibility.

To begin with, we set up an additional service in our module. To achieve this, we prepare the src/Helloworld/Service/LoggingService.php file with the following contents:

```php
<?php
namespace Helloworld\Service;

class LoggingService
{
	public function onGetGreeting()
	{
		// Logging-Implementierung
	}
}
```

Listing 7.10

We will ignore the specific implementation of logging at this time because we are primarily considering the interaction of different services via the event system. We want to invoke onGetGreeting() as soon as the getGreeting event of the GreetingService occurs.

In addition, we ensure that the service is known to the system. To achieve this, we add the appropriate class as an invocable to our module's Module.php in the getServiceConfig() method.

```php
<?php
// [..]
'invokables' => array(
        'loggingService' => 'Helloworld\Service\LoggingService'
)
// [..]
```

Listing 7.11

From now on the `LoggingService` can thus be requested via the `loggingService` key in the `ServiceManager`. So far so good. Now, we must additionally ensure that the `GreetingService` can trigger an event on the basis of which the `onGetGreeting` method of the `LoggingService` can be run. To make the `GreetingService` available to an `EventManager` and concurrently to register the execution of the `onGetGreeting` method of the `LoggingService`, we slot a factory ahead of the `GreetingService`:

```php
<?php
namespace Helloworld\Service;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class GreetingServiceFactory implements FactoryInterface
{
        public function createService(ServiceLocatorInterface $serviceLocator)
        {
                $greetingService = new GreetingService();

                $greetingService->setEventManager(
                        $serviceLocator->get('eventManager')
                );

                $loggingService = $serviceLocator->get('loggingService');

                $greetingService->getEventManager()
                        ->attach(
                                'getGreeting',
                                array($loggingService, 'onGetGreeting')
                );

                return $greetingService;
        }
}
```

Listing 7.12

We file the factory in `src/Helloworld/Service/GreetingServiceFactory.php`; it is thus located in the same directory as the service itself. Initially, the factory generates the `GreetingService`, which it ultimately also returns. Beforehand, the factory additionally arranges for an `EventManager` for the `GreetingService` so that the `GreetingService` can now also trigger events and manage listeners. And then the `LoggingService` and its `onGetGreeting()` method are registered for the `getGreeting` event. By using a closure, we can even ensure that the $loggingService, which we directly request in the above example, is also first requested at the moment of the actual event via "lazy loading".

```php
<?php
namespace Helloworld\Service;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class GreetingServiceFactory implements FactoryInterface
{
	public function createService(ServiceLocatorInterface $serviceLocator)
	{
		$greetingService = new GreetingService();

		$greetingService->setEventManager(
			$serviceLocator->get('eventManager')
		);

		$greetingService->getEventManager()
			->attach(
				'getGreeting',
				function($e) use($serviceLocator) {
					$serviceLocator
						->get('loggingService')
						->onGetGreeting($e);
				}
		);

		return $greetingService;
	}
}
```

Listing 7.13

The advantage is obvious: the `LoggingService` is actually only the generated when it is also to be used. Indeed, the `getGreeting` event may simply never take place.

We still have to adapt the `getServiceConfig()`method of our module's `getServiceConfig()` class such that the `ServiceManager` now uses the new factory to generate the `GreetingService`. The `getServiceConfig()` then appears as follows:

```php
<?php
// [..]
public function getServiceConfig()
{
    return array(
        'factories' => array(
            'greetingService'
                                    => 'Helloworld\Service\GreetingServiceFactory'
        ),
        'invokables' => array(
            'loggingService'
                                    => 'Helloworld\Service\LoggingService'
        )
    );
}
```

Listing 7.14

We now teach the `GreetingService` how to set off the corresponding event:

```php
<?php
namespace Helloworld\Service;

use Zend\EventManager\EventManagerInterface;

class GreetingService
{
        private $eventManager;

        public function getGreeting()
        {
                $this->eventManager->trigger('getGreeting');

                if(date("H") <= 11)
                        return "Good morning, world!";
                else if (date("H") > 11 && date("H") < 17)
                        return "Hello, world!";
                else
                        return "Good evening, world!";
```

```php
	}

	public function getEventManager()
	{
		return $this->eventManager;
	}

	public function setEventManager(EventManagerInterface $em)
	{
		$this->eventManager = $em;
	}
}
```

Listing 7.15

And that was about it. If the `getGreeting()` method is invoked, the corresponding event, for which we registered our logger, will be triggered.

If we use the `SharedServiceManager`, we can simplify the `GreetingServiceFactory` even more:

```php
<?php
namespace Helloworld\Service;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class GreetingServiceFactory implements FactoryInterface
{
	public function createService(ServiceLocatorInterface $serviceLocator)
	{
		$serviceLocator
			->get('sharedEventManager')
			->attach(
				'GreetingService',
				'getGreeting',
				function($e) use($serviceLocator) {
					$serviceLocator
						->get('loggingService')
						->onGetGreeting($e);
				}
		);

		$greetingService = new GreetingService();
```

```
                return $greetingService;
        }
}
```

Listing 7.16

However, then one must ensure that—here in an exemplary manner directly in the service before the event is set off—the respective `EventManager` also feels responsible for that “identifier”. To achieve this, one used the `addIdentifiers()` method:

```php
<?php
namespace Helloworld\Service;

use Zend\EventManager\EventManagerAwareInterface;
use Zend\EventManager\EventManagerInterface;
use Zend\EventManager\Event;

class GreetingService implements EventManagerAwareInterface
{
        private $eventManager;

        public function getGreeting()
        {
                $this->eventManager->addIdentifiers('GreetingService');
                $this->eventManager->trigger('getGreeting');

                if(date("H") <= 11)
                        return "Good morning, world!";
                else if (date("H") > 11 && date("H") < 17)
                        return "Hello, world!";
                else
                        return "Good evening, world!";
        }

        public function getEventManager()
        {
                return $this->eventManager;
        }

        public function setEventManager(EventManagerInterface $em)
        {
                $this->eventManager = $em;
        }
}
```

Listing 7.17

The event object

The event trigger and the event listener communicate via the event object, which is generated by the event trigger and which is made available to the event listener on invocation. When the event is triggered with the aid of its name

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

Listing 7.18

an object of the `Zend\EventManager\Event` type is transferred to the listener, the former is automatically generated and does not provide much more than an internal data structure for the generic transfer of parameters as well as the information on the so-called "target", i.e. the place where the event itself is triggered and the designation of the event itself. If one desires to make specific data available to the listeners, this can be done with the aid of parameters:

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

Listing 7.19

On can then access the data structure in the listener via the event object's `getParams()` method. Thus, practically everything that one generally needs can already be achieved with the aid of the `Zend\EventManager\Event`. However, if one desires to work with specific designators and member variables, which one can access via getters and setters, instead of the generic data structure, one has the option, as application developer, of declaring one's own event class, which can be instantiated and filled "on the fly", as needed. Zend Framework 2 itself actively uses this mechanism for, indeed, there is a `ModulEvent`, a `ViewEvent`, and also a `MvcEvent` that, for example, allows direct access to the following objects:

```php
<?php
protected $application;
protected $request;
protected $response;
protected $result;
protected $router;
protected $routeMatch;

// [..]
```

Listing 7.20

One can best derive one's own event class from `Zend\EventManager\Event`:

```php
<?php
namespace Helloworld\Event;

use Zend\EventManager\Event;

class MyEvent extends Event
{
        private $myObject;

        public function setMyObject($myObject)
        {
                $this->myObject = $myObject;
        }

        public function getMyObject()
        {
                return $this->myObject;
        }
}
```

Listing 7.21

To register the event class, one can either make this information known beforehand and then invoke the event by using the event name

```php
<?php
$this->eventManager->setEventClass('Helloworld\Event\MyEvent');
$this->eventManager->trigger('getGreeting');
```

Listing 7.22

or one transfers the corresponding object in the scope of `trigger()`:

```php
<?php
$event = new \Helloworld\Event\MyEvent();
$event->setName('getGreeting');
$this->eventManager->trigger($event);
```

Listing 7.23

In this manner, one could also write completely distinct event classes, whose name is also already predefined:

```php
<?php
namespace Helloworld\Event;

use Zend\EventManager\Event;

class MyGetGreetingEvent extends Event
{
        private $myObject;

        public function __construct()
        {
                parent::__construct();
                $this->setName('getGreeting');
        }

        public function setMyObject($myObject)
        {
                $this->myObject = $myObject;
        }

        public function getMyObject()
        {
                return $this->myObject;
        }
}
```

Listing 7.24

The actuator of this event would then be less susceptible to errors and the code, even more compact:

```php
<?php
$event = new \Helloworld\Event\MyGetGreetingEvent();
$this->eventManager->trigger($event);
```

Listing 7.25

Modules

In addition to the `EventManager`, the “module” concept plays a decisive role in Zend Framework 2. In the request processing in this framework, the `ModuleManager` is important because it indeed insures that the activated module is always considered and loaded, i.e. that the application is fully functional.

The important thing is that the application’s modules do not form any “closed entities”. The opposite is true: The functions of the individual modules merge in the scope of module loading to form the “overall functionality” of the application. We have seen that the `ModuleManager` unites the configurations of all modules in an application-wide configuration object. This fact has several consequences, which one, as application developer, should be aware of. All “Services, which make a module available, make it available in an application-wide manner, i.e. for example “controller plugins” or “view helpers”, as well as, for example, the controllers of one module are also always available to the other modules, there is not separate `ControllerLoader` for each module.

The “Application” module

If one begins to develop one’s own application on the basis of the `ZendSkeletonApplication`, one runs into the `Application` module almost immediately. If one is aware of the above-mentioned fact that the functionality of all modules yields the overall functionality of the application and if one risks a look into what is made available by this module, one quickly becomes aware of the purpose of this “standard module”. It configures a number of services and generates several basic functions which every application must have and whose own implementation does not have to be performed. In addition to the fact that the `ZendSkeletonApplication` only provides an exemplary controller for the application’s “start page” with its route configuration, a number settings for the “view layer” are contained in the `module.config.php` of the `Application`. These are then read by the `ViewManagerFactory` or the `ViewManager` and either utilized by the latter themselves or passed on to other objects:

```php
<?php
// [..]
'view_manager' => array(
    'display_not_found_reason' => true,
    'display_exceptions' => true,
    'doctype' => 'HTML5',
    'not_found_template' => 'error/404',
    'exception_template' => 'error/index',
    'template_map' => array(
                'layout/layout'
                        => __DIR__ . '/../view/layout/layout.phtml',
                'application/index/index'
                        => __DIR__ . '/../view/application/index/index.phtml',
                'error/404'
                        => __DIR__ . '/../view/error/404.phtml',
                'error/index'
                        => __DIR__ . '/../view/error/index.phtml',
        )
)
```

Listing 8.1

- Via `display_not_found_reason`, the `RouteNotFoundStrategy` (the standard manner in which 404 errors are handled) is instructed to make the reason for the fact that a URL resulted in a 404 error available for further presentation As we saw in the previous chapters, every result of request processing is based on the "view model", which contains both the user data and the (HTML) template that are to be used for this purpose. However, when a 404 error occurs, there is generally no view model generated by a controller, simply because there was no responsible controller. The `RouteNotFoundStrategy` then ensures that a "view model" is generated so that the process described above can ever take place. The `ZendSkeletonApplication` also provides an appropriate template in `view/error/404.phtml`. The problem cause can be accessed there via `$this->reason`.
- `not_found_template` explicitly determines the template this is to be used in 404 situations. As standard, Framework searches for the `404.phtml` template. It would thus be adequate if one were to create a corresponding `404.phtml` template in the `view` directory of one of the modules. Incidentally, if one creates a `404.phtml` file in more than one module, the one in the most recently activated module is used. As you may remember, the configurations of the individual modules are merged on loading by the `ModuleManager`. The `error/404` template for 404 situation is determined in the `ZendSkeletonApplication`, which in turn indicates to a physical file by means of the following ‘template_map configuration.

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

Listing 8.2

- `display_exceptions` functions similarly to `display_not_found_reason`. In this case, the `ExceptionStrategy` is analogously instructed to make the `$this->display_exceptions` available in the appropriate "view model" and thus in the defined template. In this manner, a decision can be made there as to whether or not further details of the exception are to be presented.
- With `exception_template` the template that is implemented in exception situations—i.e., always when an exception appears somewhere in processing that is not noticed and processed by the application developer—is explicitly determined. As standard, Framework searches for the `error.phtml` template. It would thus be adequate if one were to create a corresponding

`error.phtml` template in the `view` directory of one of the modules. The `error/index` template for exceptions situations is determined in the `ZendSkeletonApplication`, which in turn indicates to a physical file by means of the following ‘template_map configuration:

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

Listing 8.3

- The “doctype view helper”, which we will look at in more detail later, is configured via `doctype`. The “doctype declaration” can be output in a template or a layout (more about this also later) via the “doctype view helper” without one’s having to add it manually. The deposited value is used by the `ViewHelperManagerFactory` to furnish the `doctype` view helper with the appropriate configuration.
- Then the template that functions as a “visual frame” is defined by means of `layout` key. As standard, Framework expects the layout template under the `layout/layout` key in the template map or in an appropriate file in the file system. If necessary, a different template could be declared under the `layout` key.

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

Listing 8.4

The “application” module is thus basically nothing special, but does reduce the initial configuration effort by the application developer. I recommend conscious further development of the “application” module and the deposition of all definitions and configurations that all or a majority of the modules have in common there. Theoretically, one can also design an application such that does not contain any modules other than “application” and in which all functions are directly realised there. In some cases this is certainly an effective procedure.

Module-dependent behaviour

Due to the fact that, after the modules have been loaded, there are no more modules in the running application, but instead there is only the application itself with all its functions and configurations, which has been created by merging the individual modules. Consequently, there is basically no possibility to configure module-dependent behaviour at a later time because no information on the currently active module is available. However, if one wants to execute a certain action, for example when the controller of a specific module is executed, one has to resort to accessing the `SharedEventManager` and a trick:

```php
<?php
namespace Helloworld;

use Zend\ModuleManager\ModuleManager;

class Module
{
        public function init(ModuleManager $moduleManager)
        {
                $sharedEvents = $moduleManager->getEventManager()
                        ->getSharedManager();

                $sharedEvents->attach(
                        __NAMESPACE__,
                        'dispatch',
                        function($e) {
                                $controller = $e->getTarget();
                                $controller->layout('layout/helloWorldLayout');
                        },
                        100
                );
        }
}
```

Listing 8.5

In this example, we configure a different layout for all pages that are generated by means of a controller action of our `Helloworld` module. To achieve this, we attach a callback function to the `dispatch` event, but only for those controllers that feel responsible for the `Helloworld` "identifier" (this value corresponds to the magic constant `__NAMESPACE__` in this case), i.e. for all the controllers of the `Helloworld` module. How exactly does this work? In this case, we have a special situation in that we desire to register a listener (in this case in the form of a callback function) with an "EventManager" that does not yet exist at the time of registration. A controller is indeed only assigned to an event manager of its own at the time of its generation. This means that we must use the `SharedEventManager` (see also previous chapter). And here is the trick: A controller in Zend Framework 2 is always configured such that—when it is derived from an `AbstractActionController`— it consults the `SharedEventManager` when its listener is alerted; among other things it declares the controller namespace (i.e. in this case "Helloworld"). And we implemented the callback in `init()` method above for just this purpose; the former is now invoked, obtains the determined controller via the `MvcEvent` object, and then stipulates another layout template with the aid of the "controller plugin". The template itself must naturally exist or it must additionally be made available via the template map. Otherwise, this results in an error.

Expressed in another way, when one registers listeners for the events of an identifier that corresponds to the namespace of a module, they are taken into consideration by the controllers of the respective module.

Installing a third-party module

On of the major achievements of Version 2 is the fact that one's own application can be extended in a simple manner without having to program it oneself. For the installation of a third-party module, a few fundamental operational steps are required and depending on the module a few manipulations may be necessary.

Sources for third-party modules

Two good sources for obtaining modules are the official ZF module page[33] and GitHub[34]. In this context, the ZF-Commons-Repository[35] should be particularly mentioned. But additional repositories can be quickly found by using GitHub's search function. A brief note at this time: not all available modules have the quality that one sets as the minimum standard for one's own code. Many modules, particularly those with very small version numbers still contain numerous bugs and security gaps. Because the functions and configurations of all modules are merged by the `ModuleManager` in the scope of module loading, previously registered services, etc. can suddenly no longer be available, for example, because they were unintentionally overwritten by other implementations. It is therefore important to use third-party modules with care and to make conscious decisions for or against using a certain module.

Installation of a third-party module

There are many options for making additional modules available for one's own application. The simplest way to manually download the module codes and to copy the source files into one's own application. In this manner, for example, one can download the Module ZfcTwig [36] and copy the contents of the archive into the `module` folder of one's own application (e.g. in a "ZfcTwig" directory). If one then remembers to activate the module in the `application.config.php` (add the value "ZfcTwig" in the `modules` section), the module is basically ready to run. However, `ZfcTwig` is a module that, in turn, still requires the PHP library "Twig[37]" in order to be ready for operation. "Twig" is a Template Engine[38] from the makers of Symfony Frameworks, one of the major rivals

[33] http://modules.zendframework.com/
[34] https://github.com
[35] https://github.com/ZF-Commons
[36] https://github.com/ZF-Commons/ZfcTwig/tarball/master
[37] http://twig.sensiolabs.org/
[38] http://de.wikipedia.org/wiki/Template_Engine

of Zend Framework. In the meantime “Twig” has slightly outstripped the old top dog “Smarty[39]” and is gladly and already frequently used. Incidentally, one can ask oneself why one needs another template language like “Twig” for the development of templates in PHP at all, especially since PHP itself is a template language. After all, the typical Zend Framework “phtml” files also consist of only HTML, PHP code and a few “view helpers” as required, and Zend Framework 2, just like Version 1, does not provide another templating engine. These questions are all very correct and very valid. Th short answer to these questions is: “No, we really do not need any additional templating system, whose syntax has to be additionally learned and whose rough edges have to be known.” Everything is good the way it is. However, there are application situations in which an additional template engine can be very helpful if one desires to prevent “shoddy work” from being implemented in the templates A pure PHP-based template provides all the options of PHP, for example access to all functions of the language core. When we prevent that and intentionally want to restrict the possibilities in the templates to a defined set of functions, a template engine can be helpful—also beyond the “syntactic sugar”, the other main reason for its use.

But let’s go back the actual problem of the dependence of the `ZfcTwig` module on the “Twig” library. The `ZfcTwig` module thus functions as a kind of “glue code” and ensures that “Twig’s” functions can be used in the context of a Zend Framework 2 application. Therefore, we can now download “Twig” next, deposit it in the `vendor` directory and configure the autoloading of “Twig” at the appropriate places. Then it will function.

Because this process is error-prone und time-consuming, it is also advisable to use “Composer[40]” for the installation of third-party modules whenever possible, as already done during the installation of `ZendSkeletonApplication` itself. If we do it in this manner, the “Composer” not only handles the download and making the `ZfcTwig` module available, but also deals with its dependencies, i.e. the “Twig” library. To achieve this, we extend the `composer.json` of our application in the `require` section:

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

Listing 8.6

and run “Composer” in the project directory again

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

“Composer” now downloads the required libraries and also sets up the autoloading for us. However, if one now takes a look into the `module` directory of one’s own application, one discovers that no module has been added there. Instead “Composer” deposits all of the libraries that it loads in the `vendor` directory as standard. In the `application.config.php`, the following section controls

[39]http://www.smarty.net/
[40]getcomposer.org

```php
<?php
// [..]
'module_paths' => array(
        './module',
    './vendor',
),
```

Listing 8.7

the paths in which the application expects installed modules. `vendor` is thus completely okay and even completely different paths can be configured at this location.

If one now opens `/sayhello`, one sees … a jumbled mass of letters. Namely, the templates that we are currently still using in our `Helloworld` application are based on PHP and are thus incomprehensible to "Twig". "Twig" indeed requires "Twig"-conform template markup. Fortunately, `ZfcTwig` provides alternative "Twig"-compatible templates for the `ZendSkeletonApplication` exemplary pages, which one can use instead of the "normal" templates of the `application` module. Subsequent to installation of `ZfcUser` by "Composer", one finds them at `vendor/zf-commons/zfc-twig/examples`. One overwrites the current templates in `module/application/view` with these files. By doing so, we have already made the layout and the templates for the error pages "Twig"-compatible. One must now also do the same with the templates that one has written oneself. Then one can invoke `/sayhello` in the accustomed manner. The templates are now processed by "Twig".

Configuring a third-party module

Other third-party modules have a structure more like that of "MVC" modules; they provide controllers, views, routes, etc. A good example of such a module is ZfcUser[41], which maps the functions of a user registration including "Log-in/Log-out" flows. We will also look at this module again in detail at a later time.

For simplicity's sake, let's imagine for a moment that the `Helloworld` module in the last chapter were a third-party module, which we had installed with "Composer". With the `Helloworld` module, the URL `/sayhello` would now thus have entered our system. But what would we do if we actually desired to make the corresponding page available under the URL `/welcome`?

Naturally, we could now open the module's `module.config.php` and make our modifications there directly. But that would mean that we had "bifurcated" the `Helloworld` code base and thus improvements, which had been made on `Helloworld` by the original author (we're pretending that it is a third-party module) could no longer be imported into our system as easily. We would overwrite our modifications every time we updated and would have to manually alter the data again after every update. That's not very optimal.

[41]https://github.com/ZF-Commons/ZfcUser

Instead, we will file our modifications in a module of our own and thus separate it from the `Helloworld` code base. In this manner, `Helloworld` retains its original structure and could be updated at any time without risk that our modifications would be lost.

Changing URLs

In order to change the URL `/sayhello` into `/welcome`, we create a new subdirectory and thus a new module in the `module` directory and name it `HelloworldMod`. In this manner, we make it clear that this is our modification of another module. There we only need to have the `Module.php` in whose `getConfig()` method we pick up on the route definition for `sayhello` and overwrite it with `/welcome`.

```
<?php
namespace HelloworldMod;

class Module
{
        public function getConfig()
        {
                return array (
                        'router' => array(
                                'routes' => array(
                                        'sayhello' => array(
                                                'type' => 'Zend\Mvc\Router\Http\Literal',
                                                'options' => array(
                                                'route'    => '/welcome'
                                                )
                                        )
                                )
                        )
                );
        }

        // [..]
}
```

Listing 8.8

Now we have to ensure that the module will be activated and to do this we extend the `application.config.php` in the `modules` section appropriately.

```php
<?php
return array(
        'modules' => array(
                'Application',
                'Helloworld',
                'HelloworldMod'
        ),
        'module_listener_options' => array(
                'config_glob_paths'    => array(
                        'config/autoload/{,*.}{global,local}.php',
                ),
                'module_paths' => array(
                        './module',
                        './vendor',
                ),
        ),
);
```

Listing 8.9

When we now invoke the URL `/sayhello`, we now receive, as expected, a `404` error. In contrast, `/welcome` now yields the accustomed page.

In this context, we make use of two mechanisms. First, the configuration of all modules is merged in the scope of their loading process such that we can quasi also make configurations "for other modules" in one of the modules. Second, the configuration of the modules is individually and successively read in. Thus, in our case, they are read in as follows: initially that of the `Application`, then that of `Helloworld`, and finally the configuration of the `HelloworldMod`. In this manner, configuration of a previous module can be overwritten by the following one, as we did for the route with the designation `sayhello`, which initially is defined by `Helloworld` and then is subsequently "reconfigured" in its `route`option from `Helloworld` to `/welcome`.

Changing views

The presentation of third-party modules can also be adapted. This is generally more important and more frequently required than the adaptation of URLs. To achieve this, the `Module` class of `HelloworldMod` is extended by the `view_manager`section and there the template of the `index` action of the `index` controllers of `Helloworld` module is pointed to the appropriately modified template in the `HelloworldMod` module.

```php
<?php
namespace HelloworldMod;

class Module
{
        public function getConfig()
        {
                return array (
                        'router' => array(
                                'routes' => array(
                                        'sayhello' => array(
                                                'type' => 'Zend\Mvc\Router\Http\Literal',
                                                'options' => array(
                                                        'route'    => '/welcome'
                                                )
                                        )
                                )
                        ),
                        'view_manager' => array(
                                'template_map' => array(
                                        'helloworld/index/index'
                                                => __DIR__ . '/view/helloworld-mod/index/index.pht
                                )
                        )
                );
        }
}
```

Listing 8.10

If we now invoke `/sayhello`, we see the rendered page under utilisation of the new template.

Controller

Concept & mode of operation

A controller's task is to process an interaction with the user interface, in our case thus a website or in a wider sense with the browser itself, respectively.

The MVC pattern (the "C" here stands for "Controller") is incidentally already fairly old. It was first used at the end of the 1970s for the realisation of user interfaces when the Smalltalk[42] programming language was still popular. However, the user interfaces as well as the processing of the interaction was generally restricted to a closed system at that time. This is different in the web. There the user interface is manifested in the browser, i.e. on the client, whereas the processing takes place on the server. The browser transmits the information on what is occurring on it or on the website which it is displaying, and how the user is interacting with it. If the user clicks on a link, the browser transmits the fact that the user has just requested another webpage to the server. On the server Framework now determines which controller in the system is stipulated for the processing of this interaction (routing) and passes the further responsibility for the generation of a response to it. Subsequently, its ensures that the representation is changed on the user's browser. Either a completely new page is loaded or, if AJAX is being used, only certain parts of the page already being displayed will be updated.

Normally, one creates a controller one one's own for every "page type" in the system. For example, in an online shop there would be an "IndexController" for the homepage, a "CategoryController" for the presentation list of offered items in a specific category and an "ItemController" for the presentation of a detail page of the offered items. In Zend Framework, a controller is further subdivided into so-called "actions". The actual processing thus does not take place in the controller itself, but in its actions instead. In this context, actions are public methods of the controller class. In addition to the above-mentioned controllers, an online shop normally provides a shopping cart, and the system would therefore probably have a "CartController". In order to present the normal interactions with a shopping cart, the "CartController" would be equipped with a number of actions, among them for example the "show action", the "add action", the "remove action", the "remove all action", etc.

An appropriate procedure is to keep the code in the controllers or actions themselves, respectively, as lean as possible. In the ideal case, a controller should only register and evaluate the interaction and then decide which lever to pull in order to generate the desired result. As a rule, it uses services, which allow accessing of databases, sessions or other information or functions, to do this.

Controller plugins

Additionally, controllers frequently use so-called "controller plugins". They encapsulate interaction-relevant code that is often required and can thus be reused by different controllers.

[42] http://de.wikipedia.org/wiki/Smalltalk

Framework comes equipped with a large number of plugins, which can be immediately used in one's own controllers. All plugins inherit the same basic class, which ensures that the respective plugin always also has access to the controller in which it is just being used. This is practical in many situations; in some of them it is indispensible, as we will see in the following.

Redirect

The redirect plugin makes it possible to transfer a client to another URL: In this context, this redirect is not performed "internally by Framework", but rather by feedback to the client, which then actively invokes the new URL. To achieve this, Framework sends a response with a 302 HTTP status code[43] and the client, in most cases the user's browser, sends a new request to the specified URL.

In this context, a target URL can be entered directly.

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

Listing 9.1

Or a URL can be generated on the basis of the routes available in the system. Thus, if we desire to forward something to the URL `http://localhost:8080/sayhello`, we can alternatively allow the URL to be generated using the route configuration:

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

Listing 9.2

In this case, `sayhello` is the name of the route which we specified in our module configuration file, `module.config.php`:

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'sayhello' => array(
            'type' => 'Zend\Mvc\Router\Http\Literal',
            'options' => array(
                'route'    => '/sayhello',
                'defaults' => array(
                    'controller' => 'Helloworld\Controller\Index',
```

[43]http://de.wikipedia.org/wiki/HTTP-Statuscode

```
                    'action'     => 'index',
                )
            )
        )
    )
)
```

Listing 9.3

But, what happens if, the URL is not a static string as in our example, but is dynamically compiled, as, for example, in `http://www.zalando.de/nike-performance-free-run-3-laufschuh-black-relfecting-silver-p` There is also a solution in this case: In addition to the name of the route, the dynamic components, which are then combined to form the URL, can be transferred as additional parameters. In a later chapter, we will again examine the routing mechanism in detail and will then also again reconsider the contents of the redirect plugin. At that time, we will see exactly how a target URL is created from the dynamic components.

The redirect plugin takes advantage of the fact that when a controller a `Response` type object returns, the otherwise customary further processing is skipped. Appropriately, no further rendering of the view occurs in this case.

In order for the redirect plugin to function properly, it is necessary that the controller in which it is used has the `MvcEvent` object because the `Router` that is required to generate a URL on the basis of the route name is obtained from it. Customarily, one always derives one's own controller from the Framework basic class `AbstractActionController`, in which a large number of controller-relevant interfaces has already been implemented and thus, for example, ensures that `MvcEvent` is available:

```php
<?php
namespace Zend\Mvc\Controller;

// use [..];

abstract class AbstractActionController implements
Dispatchable,
EventManagerAwareInterface,
InjectApplicationEventInterface,
ServiceLocatorAwareInterface
{
        // [..]
}
```

Listing 9.4

Framework's `AbstractActionController` implements the `InjectApplicationEventInterface` and thus allows the `ControllerLoader`, the service that is responsible for instantiating and invoking the controller appropriate for the route after routing, to inject the `MvcEvent` object. Incidentally, the implementation of the `EventManagerAwareInterface` ensures that an `EventManager` is available to the controller, and the implementation of the `ServiceLocatorAwareInterface` ensures that the controller can access the `ServiceManager` and thus also, the application's services.

PostRedirectGet

This plugin provides support in a special redirect problem in connection with forms that are sent via POST. If the user initiates a reload of the page after previous dispatch of a form to its confirmation page, the POST data are again transmitted to the server in most cases, and in untoward cases, undesired double purchases, bookings, etc. are made. To avoid this problem, a confirmation page, which is displayed subsequent to a POST-based transaction, should not be generated in the same step. Instead, in response to the successful POST request, the server initially sends a 301/302 HTTP status code and then forwards it to a confirmation page. The latter is then requested by the browser via a GET and thus can also be invoked several times without risk. The PostRedirectGet-Plugin ensures that one does not have to develop this mechanism oneself.

Forward

The redirect plugins realise rerouting in the client via the appropriate HTTP header, whereas the forward plugin allows the invocation of another controller from within a controller. Actually, the designation "forward" is not exactly correct. The forward plugin does not actually forward anything, but instead merely "dispatches" another controller. If one examines the controllers in Zend Framework 2 again closely, the following becomes clear: a controller is really nothing more than a class which has a`dispatch()` method at its disposal, which expects a `Request` type object and a `Response` type object; and which thus meets the [conditions for] a `DispatchableInterface`. What the respective controller does with the `Request` so that the former can subsequently return an appropriate `Response` is left up to controller itself.

Accordingly, with the forward plugin only the `dispatch()` method of a controller is run, and it returns—as is usually the case when a controller is executed—an object of the `ViewModel` type as result. Indeed, what now happens to the result is left to the controller discretion. If one desires to emulate a "controller forward", the following one-liner shows how to do it:

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

Listing 9.5

In this case, "other" allegorically represents another controller, which was also previously also made known to the system in the scope of module configuration. In this case, the `ViewModel`, which the `OtherController` generates, is returned 1-to-1 by the originally invoking controller, which no longer generates a `ViewModel` itself.

However, the forward plugin can also be used to aggregate the results of several other controllers. But we will discuss this later in depth in the “concept & mode of operation” section of the chapter on the “view” topic.

Incidentally, if one desires to invoke a specific `Action` by another controller, this can be realised as follows:

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

Listing 9.6

In this manner, other additional information can also be transferred to the invoked controller:

URL

With the aid of the URL plugin, a URL can be generated on the basis of a previously defined route und under declaration of the route designation as well as additional parameters as required:

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

Listing 9.7

Params

Allows simple access to request parameters, which would otherwise be difficult to access. Actually, the POST parameters in a controller would have to be accessed as follows:

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

Listing 9.8

By using the plugin, access can be performed somewhat more elegantly even if the routine is not much shorter:

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

Listing 9.9

If one omits the parameter designation, all of the parameters are returned in the form of an associative array:

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

Listing 9.10

The important thing is to declare the correct source. The following invocations are possible:

```php
<?php
$this->params()->fromPost();
$this->params()->fromQuery();
$this->params()->fromRoute();
$this->params()->fromFiles();
$this->params()->fromHeader();
```

Listing 9.11

Layout

Via this layout plugin, the layout to be used can be configured at any time:

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

Listing 9.12

The layout plugin is, for example, helpful if one desires to use an alternative layout in the scope of a certain action.

Flash messenger

Messages for a user can be transported over a page change via the flash messenger. Technically speaking, to achieve this, a session is generated on the server side and the respective message thus persists briefly. A message for a user can be added to a flash messenger as follows:

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

On the target side one can retrieve and display the message (or even several messages ifs required) as follows:

```php
<?php
$flashMessenger = $this->flashMessenger();

if ($flashMessenger->hasMessages()) {
    $return['messages'] = $flashMessenger->getMessages();
}
```

Listing 9.13

Writing one's own controller plugin

In Zend Framework 2, a "controller plugin" is really nothing special. Indeed, it is just a more or less "normal class", which actually only expects a "getter" and a "setter" and which is injected via the Controller in which the respective controller plugin is now being applied. However, the implementation of this method can be avoided if one derives one's own controller plugin from the `Zend\Mvc\Controller\Plugin\AbstractPlugin` class. Then one must merely fill the `__invoke()` method with life in order to be able to easily use the "Controller Plugin" in a controller.

```php
<?php

namespace Helloworld\Controller\Plugin;

use Zend\Mvc\Controller\Plugin\AbstractPlugin;

class CurrentDate extends AbstractPlugin
{
        public function __invoke()
        {
                return date('d.m.Y');
        }
}
```

Listing 9.14

This controller plugin ensures that the current date is generated (admittedly one does not really need a controller plugin to do this). The class definition is located in the `Helloworld` module under `src/Helloworld/Controller/Plugin/CurrentDate.php`. The `index` action of the `IndexController` is adapted as follows so that it uses `CurrentDate` and provides the view with this result:

```php
<?php
// [..]
public function indexAction()
{
    return new ViewModel(
        array(
            'greeting' => $this->greetingService->getGreeting(),
            'date' => $this->currentDate()
        )
    );
}
```

Listing 9.15

`CurrentDate` can be very easily invoked as if it were a method of the current controller (via `$this`). However, in order for this invocation to function, we must make `CurrentDate` known to the `ControllerPluginManager` beforehand. We can do this either in der `module.config.php` of the module or in its `Module` class:

```php
<?php
// [..]
public function getControllerPluginConfig()
{
    return array(
        'invokables' => array(
            'currentDate'
                                    => 'Helloworld\Controller\Plugin\CurrentDate'
        )
    );
}
```

Listing 9.16

Once registered, this "controller plugin" can be invoked in all controllers. Incidentally, `CurrentDate` is not restricted to the `Helloworld` module, but also can be used in controllers of other modules. Whether or not this is really appropriate certainly depends on the specific case.

Views

Concept & mode of operation

The view, also often referred to as the "presentation layer" in the scope of the MVC pattern, is responsible for the presentation of the processing results. From the moment of the successful execution of a controller (or action) up to the presentation of the final result on the user's browser, the result presentation assumes a number of different forms and is subject to a large number of transformation processes, some of them still on the server and then some of them on the client. The initial representation of the processing result, the so-called "view model", generates a controller of the application. The "view model initially does not contain any presentation information (for example HTML), but instead merely contains the "basic data structure" in the form of key value pairs:

```php
<?php
public function indexAction()
{
    return new ViewModel(
        array(
            'event' => 'Beatsteaks',
            'place' => 'Berlin',
            'date' => $this->currentDate()
        )
    );
}
```

Listing 10.1

Alternatively, simply a PHP array can also be returned.

```php
<?php
// [..]
public function indexAction()
{
    return array(
        'event' => 'Beatsteaks',
        'place' => 'Berlin',
        'date' => $this->currentDate()
    );
}
```

Listing 10.2

Via Framework's `CreateViewModelListener`, which reacts to the `dispatch` result of an `ActionControllers` as standard, the generated array is subsequently, automatically transformed into a `ViewModel`.

However, the `ViewModel` is more than just the container for the payload data. It also contains the information on the template with which the data are later to be united. This information is also retrospectively added by Framework and the `InjectViewModelListener`; this also occurs in the scope of the `dispatch` event of an `ActionController`.

`ViewModels` can also be nested. This fact is also very helpful in practical application and is appropriate for distributing the creation of a single page across several controllers. The "Forward" controller plugin, which we are already familiar with, can be used to do this, i.e. in order to not only run a single controller or action, but indeed an entire sequence of controllers or actions, respectively, within the scope of request processing as required. In this manner, a number of `ViewModels` can be generated and processed at the same time.

```php
<?php
// [..]
public function indexAction()
{
    $widget = $this->forward()
                ->dispatch('Helloworld\Controller\Widget');

    $page = new ViewModel(
        array(
            'greeting' => $this->greetingService->getGreeting(),
            'date' => $this->currentDate()
        )
    );

    $page->addChild($widget, 'widgetContent');
    return $page;
}
```

Listing 10.3

In this `indexAction` we initially insure that the `WidgetController` (or its `indexAction`, respectively) is run. The returned `ViewModel` is cached in the `$widget` variable. In `$page` saving, the reference is cached in the actual `ViewModel` of the `indexAction`. However, before it is returned, `addChild` attaches the generated "view model" of the `WidgetController` to this, for simplicity's sake let's call it, "primary View Model". In this manner, one can access the rendering result of the view model of the `WidgetController` via the `widgetContent` key:

```
// [..]
<sidebar>
        <?php echo $this->widgetContent ?>
</sidebar>
// [..]
```

Listing 10.4

Layouts

If one knows how to nest “view models”, one can quickly deduce the mode of operation of layouts. But perhaps we should first take a step backwards: The idea behind layouts is to acquire HTML code, which is required by many or even all controllers and actions (for the present we’ll reduce it to this for simplicity’s sake; but, of course, data structures beyond HTML can also be generated with Zend Framework 2). These include META tags, the basic HTML framework, references to CSS files and the like.

In a technical sense, a layout is really nothing more than a `ViewModel` that references a `ViewModel`, which was generated by a controller action, as “child” —just as in the example given above the `ViewModel` of a controller action referenced that of another controller action. To access the controller action’s `ViewModel` within the layout template, Framework automatically registers the `content` key ; just as we manually generated the `widgetContent` key in the previous example. Thus, in the layout template the result of a controller action can be accessed in the following manner:

```
<html>
<head>
<title>Meine Seite</title>
</head>
<body>
<?php echo $this->content; ?>
</body>
</html>
```

Listing 10.5

The layout template that is automatically consulted by Framework can be controlled via the controller plugin, which was discussed in the previous chapter, or instead via a “view helper”, which we will also examine in more detail in the following.

View Helper

It is frequently necessary to further process the `ViewModel’s` data in the course of rendering or to generate additional data. Thus, for example, in the context of subjects such as “navigation”, “META tags”, etc. there are many recurrent view-related tasks which one can master once and then make re-usable in the form of “view helpers”. As standard, Zend Framework 2 provides a large number of ready-to-use view helpers, which one can immediately use. In addition, one can write one’s own view helpers.

Writing a view helper of one’s own

In Zend Framework 2, a view helper is really nothing special; it is just a more or less “normal class”, which actually only expects a “getter” and a “setter” and which is injected via the view in which the respective “view helper” is now being applied. However, one can avoid implementing these methods when one derives one’s own view helpers from the `Zend\View\Helper\AbstractHelper` class. Then one must merely fill the `__invoke()` method with life in order to be able to easily use the view helper.

```php
<?php

namespace Helloworld\View\Helper;

use Zend\View\Helper\AbstractHelper;

class DisplayCurrentDate extends AbstractHelper
{
        public function __invoke()
        {
                return date('d.m.Y');
        }
}
```

Listing 10.6

This “view helper” ensures that the current date can be output in a view (admittedly one does not really need a view helper to do this). The class definition is located in the `Helloworld` module under `src/Helloworld/View/Helper/DisplayCurrentDate.php`. The `index.phtml` file, i.e. the view of the `index` action in the `Index` controller, is adapted as follows and now uses this view helper:

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

Listing 10.7

`DisplayCurrentDate` can be very easily invoked as if it were a method of the current view (via `$this`). However, in order for this invocation to function, we must make `DisplayCurrentDate` known to the `ViewHelperManager` beforehand. We can do this either in der `module.config.php` of the module or in its `Module` class:

```php
<?php
public function getViewHelperConfig()
{
    return array(
        'invokables' => array(
            'displayCurrentDate'
                                => 'Helloworld\View\Helper\DisplayCurrentDate'
        )
    );
}
```

Listing 10.8

or

```php
<?php
'view_helpers' => array(
    'invokables' => array(
        'displayCurrentDate'
                        => 'Helloworld\View\Helper\DisplayCurrentDate'
    )
)
// [..]
```

Listing 10.9

Once registered, this view helper can be invoked in all views. Incidentally, `DisplayCurrentDate` is not restricted to the `Helloworld` module, but also can be used in the views of other modules. Whether or not this is really appropriate certainly depends on the specific case.

Model

What exactly is the “model” of an application in reality? View and controller are relatively clearly defined with respect to their contents and are well formulated in their form and function in Framework, whereas this is not as obvious for model. This is primarily due to the fact that a model can take on a very different structure depending on the application type and the situation. For example, we are already acquainted with the view model. Is this the model that is encapsulated in “MVC”? No, but it does have some similarity with the type of model that is to be considered.

Let’s approach this initially from the definition. Wikipedia[44] defines a model as follows:

1. Representation – A model is always a model of something, namely figure, representation of a natural or artificial original, which itself can also be a model.
2. Abridgement – A model generally does not include all the attributes of the original, but only those which appear relevant to the modeller or model user.
3. Pragmatism – Models do not explicitly correspond per se to their originals. They fulfil their replacement function a) for certain subjects (for whom?), b) within a certain interval of time (when?) and c) under restriction to certain conceptual or physical operations (what for?).

A model is thus always a simplified representation of reality, which at the appropriate time is adequate for the respective application purpose. Thus, the view model is a simplified representation of a webpage, restricted to the data to be displayed, information on the status of certain action elements (button active or inactive, box open or closed, etc.) as well as the reference to a template with which these data are to be subsequently linked. In the scope of the rendering, the view model becomes a further, if you will “more highly developed” and a representation of reality that is closer to reality: The fully generated markup is used for the subsequent representation of a website. This is returned to the calling program, and the browser develops another model, the DOM[45], from it. On the basis of the “Document Object Model (DOM)”, which appropriately already bears the term “model” in its name, coloured pixels appear magically on the screen in a concluding step. And even this result is actually again only a model. And even this explanation itself is only a model because, as I see it, I have omitted irrelevant information and details at this time.

But before this discussion becomes excessively philosophical, we should now turn to the “model” of “MVC”. In a stricter definition, this model is the image of the reality touched by the application, i.e. the specialised domain which everything depends on. For example, the image of e-commerce processes in online shops or marketplaces, the management of customers, contacts and incidents in CRM systems or documents, authors and rubrics in CMS systems. And those are just a few of the possible examples! This specialised model is thus essentially completely application-specific and ultimately cannot be generically defined with any more precision. Accordingly, the structure of the specialised model and its technical expression is not specified in a certain manner. If one looks

[44]http://de.wikipedia.org/wiki/Modell
[45]http://de.wikipedia.org/wiki/Document_Object_Model

for standards in the field of “enterprise applications” or typical web applications, for which Zend Framework 2 is indeed primarily intended, the definition of the Domain-Driven Design[46] presents itself. There, the following units of an application apply to a specialised model:

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

This is not intended to be an introduction into the world of “Domain-Driven Design”, particularly not, because it only represents one way of realising a model of one’s own application, and there for does not possess any universal validity. Nevertheless, some constructs of “Domain-Driven Design” correlate with those of Zend Framework, and thus it is worthwhile to take a closer look.

Entities, repositories & value objects

An “entity” represents a definite object in a system, for example a customer or an order in a shop system. As a rule, definitude is achieved by using IDs, for example a definite customer number. Sometimes it is also possible to combine an object’s characteristics such that unambiguity exists. However, this is, e.g., extremely difficult to achieve otherwise for natural persons. Thus, for example, in a city like Berlin, Munich or Hamburg there are several people who have exactly the same name. Some of them additionally even have their birthdays on the same day. However, these are conceptionally different “entities”, which one must absolutely distinguish between.

The designation entity also again appears particularly in connection with so-called “persistence”, i.e. storage of such objects in databases such that they can survive a request. Thus, the ORM System Doctrine 2[47] uses the designation “entity”, for example, for all objects that are stored and administered in a database.

A “repository” makes it possible to access the entities stored in a database. Depending on the implementation, a repository acts as a container for the SQL request for a specific entity type. However, sometimes it is also more, as we will soon see.

In contrast to an “entity”, a “value object” has no identity of its own. For example, two instances of `\DateTime` class, the PHP standard object for the representation of data and time, which were generated with the identical time stamp and thus have the same characteristics, are simply identical.

[46]http://de.wikipedia.org/wiki/Domain-Driven_Design

[47]doctrine-project.org

Differentiation is conceptionally unnecessary. As a rule, there is no need for a value object to have persistence, although this would also be fundamentally conceivable from a technical point of view.

An entity or even a value object can be represented by means of a simple class:

```php
<?php
class User
{
        private $name;
        private $email;
        private $password;

        public function setEmail($email)
        {
                $this->email = $email;
        }

        public function getEmail()
        {
                return $this->email;
        }

        public function setName($name)
        {
                $this->name = $name;
        }

        public function getName()
        {
                return $this->name;
        }

        public function setPassword($password)
        {
                $this->password = $password;
        }

        public function getPassword()
        {
                return $this->password;
        }
}
```

Listing 11.1

As we will see later in the scope of `Zend\Db` and `Zend\Form`, entities, but also value objects that are represented in this way, can be extremely useful at different places in the application.

Business services & factories

We have already discussed services and factories. A service can be generated by a factory—or by `Zend\Di`, as we will see in the next chapter. Functions that refer to the technical domain, but cannot be unequivocally classified as entities or should not be classified as such, are referred to as "business services". In a shop system, one could, for example, define a "CurrencyConversion" service, which is given a value object with the properties "value" and "currency" together with the information concerning the currency into which the conversion is to be preformed. The conversion result is returned in the form of a new value object. Essentially, it would also be possible to understand the conversion as part of a value object, and to implement the function in the appropriate class—let's call it "price" (which is defined on the basis of "value" and "currency) in this context. However, the corresponding value object must then have cognizance of the currencies supported by the system, which have presumably been deposited as entities in a database. Whether one really wants this link is questionable.

The function thus concerns not only one type of object in the system, but instead several simultaneously. The "CurrencyConversionService" would therefore be a good candidate for representation as "business service" and would therefore also be classed as a model of the application.

So-called "technical services" or "infrastructure services", which provide a non-specialised service, for example the physical dispatching of emails or SMS messages, logging functions, etc. As a rule, they are less concerned with the specialised objects of the application and do not provide any specialised function themselves. Zend Framework 2 already provides a large number of "technical services", which can be used independently of the respective expertise. Thus, in contrast to the "business services", "technical services" are not considered to be a model of the application.

Business events

In the previous chapter, we already encountered the event system and Framework‘s many events in the scope of request processing, for example: `route`, `dispatch` und `render`. With `getGreeting` we have already conceived an event of our own that was not of technical-functional nature, but instead primarily aids in keeping the business processes within the application flexible. Thus, we can adapt workflows, add or remove actions, and change the sequence of execution at any time. In addition, these specialised events are also part of the model of the application. In a shop system, for example, typical events would be, e.g., placing an item in the shopping cart, the concluded checkout or the registration of a received return on the appropriate administration console by the shop employees.

Routing

Introduction

Behind the mechanics of the “routing” is the fundamental idea that it is no longer necessary to be able assign a URL exactly to an existing (PHP) file. Instead, URLs can be freely selected and the appropriate processing logic—generally a controller and an action—are linked to them. Not just since the discipline of search engine optimisation[48] (SEO) exists and the online marketer can now ensure that the application’s URLs are also designed for “those who speak Google” and contain the correct free words, is one happy about the acquired flexibility in the formulation of URLs. Particularly for localised applications, in which URLs are to be generated in different languages, one quickly reaches one’s limits if one does not have sophisticated routing. Zend Framework 2, as was already the case for its predecessor, provides a very high-performance and flexible “routing” solution, which has been completely reprogrammed, performs better than before and also has been more coherently conceived.

Incidentally, matching a URL to a controller is already a special case because it does not necessarily have to be a URL that is consulted as the initial value for matching. As we will see later in the scope of `Zend\Console`, freely definable “commands” can also be assigned to controllers. This is, e.g., very practical for cron jobs. We will go into more detail about this later. For simplicity’s sake, I will stick with the URL as initial value in my further explanations of the subject of routing, even when these statements have general validity and are not fundamentally restricted to URLs.

Definition of routes

The specific mapping of a URL to a controller is designated as a “route”. If the URL “X” is requested, the controller performs “Y” and its action, “Z”. A simple route definition in the scope of `module.config.php` looks like this:

```
<?php
// [..]
'router' => array(
    'routes' => array(
        'sayhello' => array(
            'type' => 'Zend\Mvc\Router\Http\Literal',
            'options' => array(
                'route'    => '/sayhello',
                'defaults' => array(
                    'controller' => 'helloworld-index-controller',
```

[48]http://de.wikipedia.org/wiki/Suchmaschinenoptimierung

```
                        'action'      => 'index',
                    )
                )
            )
        )
)
// [..]
```

Listing 12.1

When the path of the invoked URL is `/sayhello`, the `helloworld index controller` and its `index` action are run. Not much more, but also no less. A fairly complex application will bring along a large number of such definitions. We will go into more detail later.

In this case, the `helloworld-index-controller` key was defined as an alias for the actual controller in the scope of the`di` configuration (there will be more information on `Zend\Di` further on in this book):

```
<?php
// [..]
'alias' => array(
'helloworld-index-controller'
        => 'Helloworld\Controller\IndexController'
)
```

Listing 12.2

The following occurs in Framework: The router accepts the `Request` and "reads" through the list of all deposited "routes". This occurs either in the form of a "stack" (i.e. the route that was added last is the first one checked for correspondence, and the first one, last) or the routes are deposited in tree form. In any case, correspondence results in the generation and return of an object of the `RouteMatch` type, and the work of the `Routers` is finished. The values transferred into the `RouteMatch` object by the `Router, among others the controller configured for this route, are then evaluated in the further processing of the` Request`. This is how the` Dispatcher‘ knows which controller it is to be instantiated.

Matching test

Framework provides a number of options for defining simple and/or complex, for example nested, matching rules.

Checking the path

We have already seen the simplest variant, the so-called `Literal` route, above. A character string that has to correspond exactly to the path of the requested URL is defined. If this is the case, the route functions and the necessary levers are moved on the basis of its configuration.

The `Regex` route, in which the path of the URL must fit a regular expression, is a more flexible option. Let's take a concrete example again: An online shoe shop could use, for example, the following URL path for detail pages, whereby the front part embodies the "slug" of the product designation, whereas the number following it represents the item number: `/converse-as-ox-can-sneaker-black/373682726` . A Regex route that would work for this type of URL looks like this:

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'detailPage' => array(
            'type' => 'Zend\Mvc\Router\Http\Regex',
            'options' => array(
                'regex' => '/(?<slug>[a-zA-Z0-9_-]+)/(?<id>[0-9]+)',
                'spec'  => '/%slug%/%id%',
                'defaults' => array(
                    'controller' => 'helloworld-index-controller',
                    'action'     => 'index',
                )
            )
        )
    )
)
// [..]
```

Listing 12.3

The controller can then access the value for the "slug", for example, as follows:

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

Listing 12.4

If we now take another exact look at the regular expression `/(?<slug>[a-zA-Z0-9_-]+)/(?<id>[0-9]+)`: The path has to begin with a "/". Subsequently, any arbitrary number of digits, letters, the underline and the hyphen can follow, but at least one "plus sign" (+) must occur at this location. Everything that is located between the front "/" and the rear "/", is termed the "slug" and is accordingly subsequently made available via the `RouteMatch` object. After the mandatory second slash ("/"), one or more digits between `0` and `9` may follow. This number is termed the "id" and is also made available in the `RouteMatch` object.

We'll take another look at the "spec" object later. It is used for the "return trip", i.e. the generation of a URL on the basis of just this route.

As an alternative to this, one could also have met the challenge by means of a `Segment` route:

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'detailPage' => array(
            'type' => 'Zend\Mvc\Router\Http\Segment',
            'options' => array(
                'route'    => '/:slug/:id',
                'defaults' => array(
                    'controller' => 'helloworld-index-controller',
                    'action'     => 'index',
                )
            )
        )
    )
)
// [..]
```

Listing 12.5

We can dispense with the "spec" option in this case because it results from the route. In addition, optional segments can be defined when one places them in brackets, e.g. "[/:id]". The controller can then again access the value for the slug, for example, as follows:

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

Listing 12.6

The `Segment` route can also be realised by means of checks base on regular expressions when one incorporates "constraints", but in this case of the individual segments:

```php
<?php
// [..]
'constraints' => array(
        'slug' => '[a-zA-Z0-9_-]+',
        'id'     => '[0-9]+'
)
// [..]
```

Listing 12.7

Checking hostname, protocol, etc.

With the aid of a `Hostname` route, it is also possible to check for a hostname.

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'detailPage' => array(
            'type' => 'Zend\Mvc\Router\Http\Hostname',
            'options' => array(
                'route'    => 'blog.meinedomain.de',
                'defaults' => array(
                    'controller' => 'helloworld-index-controller',
                    'action'     => 'index',
                )
            )
        )
    )
)
// [..]
```

Listing 12.8

If one has drawn up this rule, the path statement in the URL no longer plays any role at all. Every URL that contains the hostname `blog.meinedomain.de` will be mapped to the specified controller.

Via the `Scheme` route, one can check wither "https" is being used—very practical when one wants to make certain contents, for example the "customer account", only accessible via https.

Via the `Method` route, it is possible to use the "mode" in which the HTTP request was sent (such as "POST", "GET" or "PUT") as the basis for the check. With the aid of this route, REST[49], it is possible, e.g., to structure web services (However, as we will see later, with the `AbstractRestfulController` there is a much better solution.) or to separate form display and processing in an elegant manner:

[49]http://de.wikipedia.org/wiki/Representational_State_Transfer

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'blog' => array(
            'type' => 'Zend\Mvc\Router\Http\Literal',
            'options' => array(
                'route'    => '/contactform',
            ),
            'child_routes'  => array(
                'formShow' => array(
                    'type'    => 'method',
                    'options' => array(
                        'verb' => 'get',
                        'defaults' => array(
                            'controller' => 'form-controller',
                            'action'     => 'show',
                        )
                    )
                ),
                'formProcess' => array(
                    'type'    => 'method',
                    'options' => array(
                        'verb' => 'post',
                        'defaults' => array(
                            'controller' => 'form-controller',
                            'action'     => 'process',
                        )
                    )
                )
            )
        )
    )
)
// [..]
```

Listing 12.9

In this case, whenever one is dealing with a GET request on the URL `/contactform`, the `show` action of the `form-controller` will be run and when one is dealing with a POST request, the `process` action takes over. Otherwise, one would have to implement a ungainly “isPost()” logic in the action itself in order to be able to differentiate between the initial representation of the form (GET) and its processing (POST) (in the practice section we will develop a lovely solution for the processing of forms on the basis of special controller types).

Combining rules

Some rules make no sense when one considers them out of context. To be able to check for the hostname without having to consider the path information in any way is usually very helpful. For this reason, rules can be combined with one another in the form of trees, as one could already see in the last example. Let's consider another example: We're running an online shop under `www.meinedomain.de` and the corresponding blog under `blog.meinedomain.de`. The most recent test report is to be displayed under the URL `blog.meinedomain.de/testberichte`; on the other hand, the URL `www.meinedomain.de/testberichte` does no exist, this must thus generate a `404` error. To achieve this, a `Hostname` route and a `Literal` route can be combined:

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'blog' => array(
            'type' => 'Zend\Mvc\Router\Http\Hostname',
            'options' => array(
                'route'    => 'blog.meinedomain.de',
                'defaults' => array(
                    'controller' => 'helloworld-index-controller',
                    'action'     => 'index',
                )
            ),
            'child_routes'  => array(
                'tests' => array(
                    'type'    => 'literal',
                    'options' => array(
                        'route'    => '/testberichte',
                        'defaults' => array(
                            'controller' => 'helloworld-index-controller',
                            'action'     => 'tests',
                        )
                    )
                )
            )
        )
    )
)
// [..]
```

Listing 12.10

Only when the URL path is `/testberichte` and at the same time the hostname is `blog.meinedomain.de`, does this route become effective. This is defined in the `child_routes` section. A `child_route` can, in turn, again have its own `child_routes` , such that an entire tree structure is created. If there are several `child_routes` on the same "level", they are to be considered as alternatives:

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'blog' => array(
            'type' => 'Zend\Mvc\Router\Http\Hostname',
            'options' => array(
                'route'    => 'blog.meinedomain.de',
                'defaults' => array(
                    'controller' => 'helloworld-index-controller',
                    'action'     => 'index',
                )
            ),
            'child_routes'  => array(
                'tests' => array(
                    'type'    => 'literal',
                    'options' => array(
                        'route'    => '/testberichte',
                        'defaults' => array(
                            'controller' => 'tests-controller',
                            'action'     => 'show',
                        )
                    )
                ),
                'testArchive' => array(
                    'type'    => 'literal',
                    'options' => array(
                        'route'    => '/testberichte/archiv',
                        'defaults' => array(
                            'controller' => 'tests-controller',
                            'action'     => 'archive',
                        )
                    )
                )
            )
        )
    )
)
// [..]
```

Listing 12.11

In this case, the routes for the URL paths `/testberichte` and `/testberichte/archiv` have been defined indecently of each other; however, both proceed under the assumption that the hostname is `blog.meinedomain.de`.

Generation of a URL

If one desires to generate a link in an application that refers to an internal page, which also can be reached via a defined route, one can generate the "href" attribute that is required for the URL in a simple manner. The advantage in comparison to the manual generation of the URL is that one can subsequently adapt the form of a URL at a central location without having repeated the entire application and individually adapt every link.

Let's take a look at the (Regex) route definition again.

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'detailPage' => array(
            'type' => 'Zend\Mvc\Router\Http\Regex',
            'options' => array(
                'regex'    => '/(?<slug>[a-zA-Z0-9_-]+)/(?<id>[0-9]+)',
                'spec'     => '/%slug%/%id%',
                'defaults' => array(
                    'controller' => 'helloworld-index-controller',
                    'action'     => 'index',
                )
            )
        )
    )
)
// [..]
```

Listing 12.12

The interesting part of the generation of a URL on the basis of this route definition is the "spec". That's where one defines how a URL for this page is schematically composed. Once it has been defined, a URL can be generated via the `url` controller plugin:

```php
<?php
//[..]
$href = $this->url()
        ->fromRoute(
        'detailPage',
        array("slug" => "adidas-samba-sneaker", "id" => 34578347)
);
```

Listing 12.13

In this context, it is important to use the correct name for the route and consign all the required URL components.

Standard routing

“Standard-Routing”, which provides some comfort at the cost of flexibility, has been known and appreciated since Zend Framework Version 1. “Standard routing” ensures that the path in the URL is mapped onto a module, a controller and an action as standard. The `/blog/entry/add` URL would thus initiate the `Add` action in the `Entry` controller of the `Blog` module.

This standard routing was “baked” (hardcoded) into the MVC mechanics of Framework Version 1, this is no longer the case in Version 2: Standard routing in this form actually does not exist any more, but it can be effortlessly emulated if one read the previous sections attentively. We enable standard routing for the `Helloworld` module via the following definition:

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'helloworld' => array(
            'type'    => 'Literal',
            'options' => array(
                'route'    => '/helloworld',
                'defaults' => array(
                    '__NAMESPACE__' => 'Helloworld\Controller',
                    'controller'    => 'Index',
                    'action'        => 'index',
                ),
            ),
            'may_terminate' => true,
            'child_routes' => array(
                'default' => array(
                    'type'    => 'Segment',
                    'options' => array(
                        'route'    => '/[:controller[/:action]]',
                        'constraints' => array(
                            'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
                            'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
                        ),
                        'defaults' => array(
                        )
                    )
                )
            )
        )
    )
)
// [..]
```

Listing 12.14

However, the corresponding controller must still be initially made known in the system (that was not the case in Framework Version 1):

```
<?php
// [..]
'controllers' => array(
    'invokables' => array(
        'Helloworld\Controller\Widget'
                        => 'Helloworld\Controller\WidgetController',
        'Helloworld\Controller\Index'
                        => 'Helloworld\Controller\IndexController'
    )
 )
```

Listing 12.15

In this manner, the URLs `/helloworld` und `/helloworld/widget/index`, for example, can now be invoked.

Here are two more notes on route definition: The `may_terminate` configuration informs the `Router`, which indeed evaluates the rules, that even `/helloworld` alone, considered individually, represents a valid route and that it is not absolutely necessary to also consult the definitions of `child_routes`. However, if one omits `may_terminate` (and thus sets it implicitly to `false`), `/helloworld` would no longer function, but only the `/helloworld/widget/index` would (if we stick to the above-mentioned example in this case). And with the aid of the `'__NAMESPACE__'` option, the fully qualified class name is used for the controller that was cut out of the URL such that it fits the deposited controller configuration. Otherwise, for example, there would be an unsuccessful search for a controller, which is known under the name “Widget”, for the `/helloworld/widget/index` URL. However, its correct name, `which was automatically correctly generated in this manner, is indeed just` Helloworld\Controller\Widget.

Creative routing: A/B tests

With a bit of creativity, routing can be used for purposes which one would not initially think of. Thus, one can, for example, route part of a URL’s traffic, in this case (theoretically) half of it, onto an alternative implementation.

```php
<?php
// [..]
'router' => array(
    'routes' => array(
        'detailPage' => array(
            'type' => 'Zend\Mvc\Router\Http\Literal',
            'options' => array(
                'route'    => '/beispielseite',
                'defaults' => array(
                    'controller' => 'helloworld-index-controller',
                    'action'     => rand(0,1) ? 'original' : 'variation'
                )
            ),
        )
    )
)
// [..]
```

Listing 12.16

In this case, there must now be an `original` action und a `variation` action in the `helloworld-index-controller`, which on the one hand contains the initial version of a page and on the other hand a hypothetically improved variant of this page. The latter version should prove its superiority in the scope of an A/B test[50], for example, in the conversion rate[51]. In collaboration with a Webtracking[52] tool, such as the free Google Analytics[53], even complex A/B test scenarios can be set up and evaluated.

[50] http://de.wikipedia.org/wiki/A/B-Test
[51] http://de.wikipedia.org/wiki/Konversion_(Marketing)
[52] http://de.wikipedia.org/wiki/Web_Analytics
[53] http://www.google.com/intl/de/analytics/

Dependency injection

Introduction

The idea behind the "dependency injection (DI)" is the following: An object does not procure additional objects, which it needs, but instead they are given to it from an outside source, thus in a manner of speaking they are "injected". The largest advantage of this procedure is the fact that the dependent object itself must no longer know exactly what it is dependent on. The information on this dependency is extracted from the dependent object and managed separately. This allows the application developer to consign alternative implementations to the dependent object in certain situations, for example, when conducting unit texts.

With the use of our `GreetingServiceFactory` we have thus already practiced "dependency injection" because we extracted the information concerning the dependency of the `GreetingService` on the `LoggingService` from the `GreetingService` itself. If we once again disregard the code for event triggering and processing, the code is represented as follows: The `GreetingServiceFactory`, which ensures that the dependence of the `GreetingService` on the `LoggingService` is resolved, precedes the `GreetingService`:

```php
<?php
namespace Helloworld\Service;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class GreetingServiceFactory implements FactoryInterface
{
        public function createService(ServiceLocatorInterface $serviceLocator)
        {
                $greetingService = new GreetingService();

                $greetingService->setLoggingService(
                        $serviceLocator->get('loggingService')
                );

                return $greetingService;
        }
}
```

Listing 13.1

The `GreetingService` simply accesses the injected `LoggingService` without having considered that it already has the `LoggingService` at its disposal. In this context, it banks on the fact that just this service was made available before it was to be used, regardless of who provided it:

```php
<?php
namespace Helloworld\Service;

class GreetingService
{
        private $loggingService;

        public function getGreeting()
        {
                $this->loggingService->log("getGreeting ausgefuehrt!");

                if(date("H") <= 11)
                        return "Good morning, world!";
                else if (date("H") > 11 && date("H") < 17)
                        return "Hello, world!";
                else
                        return "Good evening, world!";
        }

        public function setLoggingService($loggingService)
        {
                return $this->loggingService = $loggingService;
        }

        public function getLoggingService()
        {
                return $this->loggingService;
        }
}
```

Listing 13.2

The `LoggingService` then accordingly carries out its work:

```php
<?php
namespace Helloworld\Service;

class LoggingService
{
        public function log($str)
        {
                // code for logging
        }
}
```

Listing 13.3

When we now want to avoid that, in the scope of the unit tests, we clog up our log file with many "spurious" accesses, we can inject a `FakeLoggingService` into the `GreetingService` during the execution of the test, which on the one hand allows the `GreetingService` to perform its work as usual without problems and on the other hand to subsequently simply discard the useless logs.

```php
<?php
namespace Helloworld\Service;

class FakeLoggingService
{
        public function log($str)
        {
                return;
        }
}
```

Listing 13.4

We can then inject the `FakeLoggingService` during the test:

```php
<?php
namespace Helloworld\Service;

class GreetingService extends \PHPUnit_Framework_TestCase
{
        public function testGetGreeting()
        {
                $greetingService = new GreetingService();
                $fakeLoggingService = new FakeLoggingService();
                $greetingService->setLoggingService($fakeLoggingService)
                $result = $greetingService->getGreeting();
                $greetingSrv = $serviceLocator$this->assertEquals(/* [..] */);
        }
}
```

Listing 13.5

As a matter of form and to ensure that the `GreetingService` is really always provided with a "logging-like service", an interface can be employed:

```
<?php
namespace Helloworld\Service;

interface LoggingServiceInterface
{
        public function log($str);
}
```

Listing 13.6

The `LoggingServiceInterface` would then be implemented by both the "real" `LoggingService`

```
<?php
namespace Helloworld\Service;

class LoggingService implements LoggingServiceInterface
{
        public function log($str)
        {
                // code for logging
        }
}
```

Listing 13.7

and the "fake implementation":

```php
<?php
namespace Helloworld\Service;

class FakeLoggingService implements LoggingServiceInterface
{
        public function log($str)
        {
                return;
        }
}
```

Listing 13.8

If an appropriate “setter” is used, a reference to a type can also be placed in the `GreetingService` in order to force the PHP interpreter to set an object of the right type:

```php
<?php
// [..]
public function setLoggingService(LoggingServiceInterface $loggingService)
{
    return $this->loggingService = $loggingService;
}
// [..]
```

Listing 13.9

Incidentally, the use of the native `date` function in the `GreetingService` makes unit testing extremely difficult. In the following chapter, when we examine unit testing in detail, we will consider a possible solution for this specific example.

Zend\Log

Incidentally, as we will soon see that, as application developers, we do not have to expend the effort to program a logging functionality ourselves at all: Zend Framework 2 already provides a very flexible logging implementation in the form of `Zend\Log`. We will go into more detail on this later.

Zend\Di for object graphs

Initially, one can best imagine `Zend\Di` as a generic factory, which one also designates as a "Dependency Injection Container". In contrast, our `GreetingServiceFactory` was very specific because we had already resolved the dependency on the `LoggingService` explicitly and manually. Alternatively, we can also charge `Zend\Di` the resolution of this dependency:.

```php
<?php
namespace Helloworld\Service;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class GreetingServiceFactory implements FactoryInterface
{
        public function createService(ServiceLocatorInterface $serviceLocator)
        {
                $di = new \Zend\Di\Di();

                $di->configure(new \Zend\Di\Config(array(
                        'definition' => array(
                                'class' => array(
                                        'Helloworld\Service\GreetingService' => array(
                                                'setLoggingService' => array(
                                                        'required' => true
                                                )
                                        )
                                )
                        ),
                        'instance' => array(
                                'preferences' => array(
                                        'Helloworld\Service\LoggingServiceInterface'
                                                => 'Helloworld\Service\LoggingService'
                                        )
                                )
                        )
                ));

                $greetingService = $di->get('Helloworld\Service\GreetingService');
                return $greetingService;
        }
}
```

Listing 13.10

Before we take a look at the actual configuration, let's first make a few additional improvements: We really do not need the `GreetingServiceFactory` any more because `Zend\Di` performs the work for us as a generic implementation of the factory. Instead, we relocate the configuration code directly in the `Module` class of the `Helloworld`-module. In the course of this, we take advantage of the fact that Framework has already automatically made an instance of `Zend\Di` available to us as a service in the `ServiceManager. We can reach this instance in two different ways. Either we request the `DependencyInjector`manually from the `ServiceManager`‘:

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

Listing 13.11

and obtain the `GreetingService` via

```php
<?php
// [..]
$di = $serviceManager->get(`DependencyInjector`);
$greetingService = $di->get('Helloworld\Service\GreetingService');
// [..]
```

Listing 13.12

or alternatively we make things a little easier for ourselves, and use the “fallback mechanism” of the `ServiceManager`: If the `ServiceManager` cannot provide a requested service because it simply does not know anything about the service in question, it asks the `DependencyInjector` once again whether it can perhaps provide help and make the respective service available. And in our case, it could. Consequently, we dispense with the `GreetingServiceFactory` entirely and shift the configuration code for `Zend\Di` directly into die `module.config.php` of the `Helloworld` module:

```php
<?php
return array(
        'di' => array(
                'definition' => array(
                        'class' => array(
                                'Helloworld\Service\GreetingService' => array(
                                        'setLoggingService' => array(
                                                'required' => true
                                        )
                                )
                        )
                ),
                'instance' => array(
                        'preferences' => array(
                                'Helloworld\Service\LoggingServiceInterface'
                                        => 'Helloworld\Service\LoggingService'
                        )
                )
        ),
        'view_manager' => array(
                'template_path_stack' => array(
                        __DIR__ . '/../view'
                )
        ),
        'router' => array(
                'routes' => array(
                        'sayhello' => array(
                                'type' => 'Zend\Mvc\Router\Http\Literal',
                                'options' => array(
                                        'route'    => '/sayhello',
                                        'defaults' => array(
                                                'controller' => 'Helloworld\Controller\Index',
                                'action'     => 'index',
                                        )
                                )
                        )
                )
        ),
        'controllers' => array(
                'factories' => array(
                        'Helloworld\Controller\Index'
                                => 'Helloworld\Controller\IndexControllerFactory'
                ),
                'invokables' => array(
                        'Helloworld\Controller\Widget'
                                => 'Helloworld\Controller\WidgetController'
                )
        ),
        'view_helpers' => array(
                'invokables' => array(
                        'displayCurrentDate'
```

```
                                => 'Helloworld\View\Helper\DisplayCurrentDate'
                )
        )
);
```

Listing 13.13

The upper section with the `di` key is new and corresponds to the code which we originally had in the `GreetingServiceFactory`. Framework searches for an entry below `di` and transfers the configuration, if present, to the `DependencyInjector`, which is indeed made available automatically. We can now remove the `getServiceConfig()` method from the `Module` class of `Helloworld`. We don't need it any more, if we make a small adjustment in the `IndexControllerFactory` and ensure that we use the fully qualified name `Helloworld\Service\GreetingService` when we request the service (up to now we had used a designation that did not correspond to the class in this case. However, to avoid a "name collision" between services of different modules, it is advisable to always use the fully qualified class name for the designation of services):

```php
<?php
namespace Helloworld\Controller;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class IndexControllerFactory implements FactoryInterface
{
        public function createService(ServiceLocatorInterface  $serviceLocator)
        {
                $ctr = new IndexController();
                $serviceLocator = $serviceLocator->getServiceLocator();

                $greetingSrv = $serviceLocator->get(
                        'Helloworld\Service\GreetingService'
                );

                $ctr->setGreetingService($greetingSrv);
                return $ctr;
        }
}
```

Listing 13.14

What happens now? The `ServiceManager` receives the request for the `GreetingService` and because it does not have a good answer at hand (we indeed removed the service configuration and moved everything into the `DependencyInjector`), it now consults the `DependencyInjector`. The latter helps the former, instantiates the `GreetingService` and in the process ensures that the `LoggingService` is on standby and that the `GreetingService` is made available. The `ServiceManager` now finally has just this operational service on hand and happily returns it to the calling program.

Here is another short piece of information: We sometimes use the term `ServiceManager` and at other times, the term `ServiceLocator`. This can be a bit confusing. The `ServiceLocator` is an interface, whereas the `ServiceManager` is its specific implementation, which Framework provides directly. Thus, the `ServiceManager` is the one that performs the actual work. As is the case at many locations in Framework, the underlying idea is that one could indeed consider replacing the `ServiceManager` with another implementation. In order that the rest of Framework and the application base on it would still function, this alternative implementation must conform to the template of the `ServiceLocator` interface. Thus, when one uses the term `ServiceLocator`, one is really always discussing its specific standard implementation, the ‘ServiceManager.

Let’s return to the details of the DI configuration mentioned above. Initially, one informs `Zend\Di` that there is a `Helloworld\Service\GreetingService` class, which has a `setLoggingService()` method at its disposal and that in any case a dependency must be made available:

```php
<?php
// [..]
'definition' => array(
        'class' => array(
                'Helloworld\Service\GreetingService' => array(
                        'setLoggingService' => array(
                        'required' => true
                )
            )
        )
)
// [..]
```

Listing 13.15

But which one is meant exactly? In this case, the `RuntimeDefinition` is brought into play: `Zend\Di` actively searches for it in the corresponding method declaration of the `GreetingService`

```php
<?php
// [..]
public function setLoggingService(LoggingServiceInterface $loggingService)
{
    return $this->loggingService = $loggingService;
}
// [..]
```

Listing 13.16

and independently locates the information as to the type of object that is to be injected in this case. If we had declared a specific class, as a sort of hint, instead of the `LoggingServiceInterface`, we would already be finished: `Zend\Di` would instantiate the appropriate class and transfer it to the `GreetingService` via `setLoggingService()` as soon as the latter was requested.

That's it! However, in our case, `Zend\Di` does not find a definite class as reference to a type, but rather a pointer to an interface. Consequently, now we must provide additional information, which actually guides `Zend\Di` to the class that is to be used in this case:

```php
<?php
// [..]
'instance' => array(
        'preferences' => array(
                'Helloworld\Service\LoggingServiceInterface'
                        => 'Helloworld\Service\LoggingService'
     )
)
// [..]
```

Listing 13.17

This configuration states the following: "When you come upon the `Helloworld\Service\LoggingServiceInterface` (and don't know hat you should do), use the `Helloworld\Service\LoggingService`". It's a simple as that! Now the `Helloworld\Service\GreetingService` can be indirectly requested via the `ServiceManager`:

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

Listing 13.18

A few chapters ago, we also created a factory of our own for the `IndexController` of the `Helloworld` module so that we could resolve its dependence on the `GreetingService`. We can also use `Zend\Di` in this manner as needed, but only when we have initially activated ("unlocked") the respective controller for loading via `Zend\Di`:

```
<?php
return array(
        'di' => array(
        'allowed_controllers' => array(
                'helloworld-index-controller'
        ),
        'definition' => array(
                'class' => array(
                        'Helloworld\Service\GreetingService' => array(
                                'setLoggingService' => array(
                                        'required' => true
                                )
                        ),
                        'Helloworld\Controller\IndexController' => array(
                                'setGreetingService' => array(
                                        'required' => true
                                )
                        )
                )
        ),
        'instance' => array(
                'preferences' => array(
                        'Helloworld\Service\LoggingServiceInterface'
                                => 'Helloworld\Service\LoggingService'
                        ),
                        'Helloworld\Service\LoggingService' => array(
                                'parameters' => array(
                                        'logfile' => __DIR__ . '/../../../data/log.txt'
                                )
                        ),
                        'alias' => array(
                                'helloworld-index-controller'
                                        => 'Helloworld\Controller\IndexController',
                        ),
                )
        ),
// [..]
);
```

Listing 13.19

The following adjustments of the `module.config.php` are necessary so that the loading of the controller via `Zend\Di` also functions:

- The `IndexControllerFactory` is removed from the `controller` section. We indeed do not desire to use it anymore, but instead want to generate the `IndexController` via `Zend\Di` in the near future.
- The `alias` section in the `instance` section below `di` has been recently added. Since no backslashes are allowed as aliases in `Zend\Di`, we use the `helloworld-index-controller` in this case. Previously, we were still able to use the `Helloworld\Controller\Index` and consequently could refer to the factory. This is unfortunately no longer possible, but that doesn't really make any difference.
- The `sayhello` route has also been adapted such that the controller alias `helloworld-index-controller` is used.
- The `allowed_controllers` section has been newly added and acquires the `helloworld-index-controller` value, i.e. the alias of the controller that we want to load via `Zend\Di`. If we forget to "whitelist" the controller in this manner, it cannot be loaded via `Zend\Di` even if all the other configurations are correct — this is a safety feature.
- Last but not least, we must now ensure that the dependence on the `GreetingService` is resolved. The factory which had ensured the resolution of the dependence on the `GreetingService` up to now has been disabled. To allow `Zend\Di` assume this task, an entry for `Helloworld\Controller\IndexController` has been added in the `di > definition > class` section, via which we inform `Zend\Di` that the `setGreetingService()` method in the `IndexController` must manditorily be invoked. When we now equip that method of the `Helloworld IndexController` with a reference to a type, `Zend\Di` can again determine for itself which class has to be injected in this case:

  <?php // [..] public function setGreetingService(\Helloworld\Service\GreetingService $service) { $this->greetingService = $service; }

  Listing 13.20

Incidentally, the `ControllerManager` has a `Zend\Di` instance of its own, via which the respective controller can now be obtained if required.

We could now thus eliminate all of our factories via `Zend\Di`. If one spins this thread further, one could basically dispense with factories completely and set up all the object graphs with all their dependencies descriptively without writing the otherwise necessary initialisation code.

`Zend\Di` should really also become the central pivotal element of `Zend\Mvc` and make its individual factories redundant. However, if one takes a close look at Framework's request processing, it becomes apparent that `Zend\Di` is practically never used and nearly all services are generated via their own factories. The `ServiceManager` and the respective factories have assumed the tasks intended for `Zend\Di` at many locations. The reasons for this were not really of a technical nature, but rather of a strategic one: to ensure that Zend Framework 2 remains easy for newcomers to learn. And `Zend\Di` unavoidably brings a great deal of hidden "Magic"" into play. Basically, the situation is also similar to one's own decision for or against `Zend\Di` and factories, respectively. Both approaches are proven methods of achieving a loose coupling of the individual players, and thus to develop a system that also remains sustainably testable, maintainable and extensible. The application developer can decide for him- or herself. By means of the above-described "fallback" mechanism, both procedures can also be combined with each other without problems and thus allow one to select the best procedure for a given situation.

Has everything suddenly become too difficult? Don't worry—it looks much worse than it really is. All one needs is a little bit of practice!

Alternative approaches to the "definition"

In addition to the `RuntimeDefinition` shown here, which makes use of the reflection mechanism to understand the code structure for the resolution of dependencies, other variants, such as the `CompilerDefinition`, which can provide the advantage of speed at the cost of convenience, can be used as required.

Zend\Di for configuration management

"Dependency injection" in general and `Zend\Di` in particular are essentially helpful in two contexts: On the one hand, complex object graphs (as sketched in an exemplary manner above) can be combined in run time without requiring the dependent objects to become active or that the hardcoded dependencies, for example, would be a hindrance of any kind to unit testing. On the other hand, individual objects can also be configured beyond their dependence with the aid of "dependency injection". Thus, to access a database or an external web service at any arbitrary location, information on the hosts as well as any available access data are required. Admittedly, the respective service of our application could now actively access the application's configuration via the `ServiceManager` and obtain the appropriate values there in a similar manner.

```php
<?php
// [..]
$config = $serviceManager->get(`Config`);
$host = $config['dbConfig']['host'];
$user = $config['dbConfig']['user'];
$pwd = $config['dbConfig']['pwd'];
// [..]
```

Listing 13.21

However, then both the dependencies on the `ServiceManager` and the information on the internal configuration structure would be hardcoded, which would be extremely disadvantageous. Here is another example: How do we best inform our `LoggingService` as to which file would be the best one for it to log into? Our best bet would be to extend the constructor of the service such that we could inject the configuration from an external source:

```php
<?php
namespace Helloworld\Service;

class LoggingService implements LoggingServiceInterface
{
        private $logfile = null;

        public function __construct($logfile)
        {
                $this->logfile = $logfile;
        }

        public function log($str)
        {
                file_put_contents($this->logfile, $str, FILE_APPEND);
        }
}
```

Listing 13.22

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

```php
<?php
// [..]
'di' => array(
    'definition' => array(
        'class' => array(
            'Helloworld\Service\GreetingService' => array(
                'setLoggingService' => array(
                    'required' => true
                )
            )
        )
    ),
    'instance' => array(
        'preferences' => array(
            'Helloworld\Service\LoggingServiceInterface'
=> 'Helloworld\Service\LoggingService'
        ),
        'Helloworld\Service\LoggingService' => array(
            'parameters' => array(
                'logfile' => __DIR__ . '/../../../data/log.txt'
            )
        )
    )
)
// [..]
```

Listing 13.23

The best thing about this is that we can carry out the configuration of the “on site” log file, i.e. physically close to the corresponding service configuration, but without having to hardcode it directly in the service class itself. In contrast to an endless configuration file, in which innumerable disjointed configuration values are strung together—as was, for example, still the case in Framework Version 1 with the `application.ini`—one now knows exactly where one has to search if something has to be altered.

Persistence with Zend\Db

Persistence, i.e. the long-term storage of data, for example in databases, has been supported by PHP for a long time. For the PDO extension, for example, there are database-specific drivers for the most common manufacturers and systems such that one nearly always effortlessly manages to connect to a database and to store data. With `Zend\Db` Zend Framework 2 provides additional support for working with databases and thus makes persistence a bit easier.

Incidentally, `Zend\Db` is not ORM[54] system and thus does not compete, for example, with Doctrine[55] or Propel[56], but instead can be understood as a "light-weight" alternative, as an additional abstraction, and be used accordingly. Anyone who uses "Doctrine" or another ORM system with thus very probably dispense with `Zend\Db`. The topic of "persistence" is unfortunately completely overloaded with extremely different, sometimes contrary designations and definitions such that it is really difficult to understand what is concealed behind an implementation. In order to be better able to classify `Zend\Db` — even if I'm running the risk of being criticised for having simplified things too greatly —let's take a brief look at the most important approaches to "persistence":

- Object Relational Mapper (ORM): The starting point for an ORM system is the mind-set that we indeed program in an object-oriented manner, but then store the data relationally. There are indeed some parallels between the two paradigms, but there are also substantial differences. One could say that an ORM system therefore attempts to make the "world that is foreign to the species", in a manner of speaking, as invisible as possible for the object-oriented thinking and acting application developer, i.e. one generates and manipulates objects and the ORM system takes over everything else that refers to databases. ORM systems frequently also provide SQL alternatives: thus, "Doctrine 2", for example, has the DQL (Doctrine Query Language), which looks a lot like SQL (This is definitely an advantage for experienced SQL users), but also has a number of additional properties to address the object-orientation as effectively as possible and thus to simplify database operations.
- Data Mapper: A "Data Mapper" is frequently considered to be the same as an ORM system, but can also support completely different "backends" than relational databases. Thus, "Doctrine 2", for example, provides the option of realising persistence in "MongoDB" or "CouchDB", both document-oriented systems. If one mentions ORM and "Doctrine 2", one should correctly speak about "Doctrine 2 ORM" because there is also "Doctrine 2 ODM" (Object Document Mapper). The main characteristic of a data mapper is the fact that it completely decouples one world (e.g. the object-oriented one) from the other world (e.g. the rational one) and functions as an intermediate "transformer" between them. Another important property of a "data mapper" is also the fact that it can store or load objects from several different sources, for example several database tables.

[54]http://de.wikipedia.org/wiki/Objektrelationale_Abbildung
[55]http://www.doctrine-project.org/
[56]http://www.propelorm.org/

- Table Data Gateway: A “Data Mapper” or an ORM system, respectively, conceptionally attempts to conceal information about the detailed data structures of “the other world” to the greatest possible extent, whereas the “Table Data Gateway” uses a completely different approach. In this case, the program “plays with the cards on the table” and generates an object for every table in the database that manages the operations around this table. The important thing is that in contrast to the “data mappers”, the focus is on a table in a relational database.
- Row Data Gateway: Whereas the “Table Data Gateway” represents a table with an object, a “row data gateway” stands as object for a row, i.e. for an “entry” if you will, of a table in the database. An important characteristic of a “row data gateway” is the fact that the respective object —in contrast to a “data mapper”—carries the requisite code to load or store itself. They are thus autonomous. In contrast, data that are downloaded from a database in a “data mapper”, frequently also termed “entities” in this context are not capable of doing this. Instead, a central “entity manager” is made available to ensure the persistence of the objects. An advantage of the “entity manager” is the fact that it operates in a so-called “unit of work” and thus can combine several similar SQL statements to form a single statement.
- Active Record: An “active record” is comparable to the “row data gateway”. The only difference, which is also rather a definitional difference, is that (in contrast to a “row data gateway”) an “active record” —in addition to the requirements of persistence—is also allowed to deal with specialist circumstances, i.e. it also contains code that not only deals with the representation of an object in a table row, but also concerns itself with the expertise of the application.
- Data Access Object (DAO): The “data access object” is difficult to define. If one considers the definition from the Core J2EE Pattern Catalogue[57], it allows access to a data source, i.e. it forms the connection to a database, and would be accordingly very “low level” and could serve as the basis for the previously-mentioned constructs, which for their part don’t even pay any attention to the definitional side. Indeed, J2E (previously: J2EE) has always had a somewhat wider conceptional basis. Thus, one quickly says that specific object types (User, Product, Category, etc.) could indeed also be obtained from different sources, such as from database systems, but also from flat files, an LDAP implementation or similar sources. Depending on the data source there would then be a respective appropriate “data access object”. However, since all persistent objects are frequently located in identical storage, “data access objects” do not play a major role or, in other words, they rather remain in the background. Except when someone says “data access object”, but really means “table data gateway”. These two terms are unfortunately indeed often used as synonyms.

Connecting to databases

The `Zend\Db\Adapter` is responsible for connecting to databases and for unifying the characteristics of the SQL implementation of individual systems, as well as the different types of database connection implemented by PHP. For, on the one hand, not all providers support all SQL standards

[57]http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

with their systems or frequently additionally provide proprietary extensions and, on the other hand, PHP has a large number of different options for database interaction at its disposal, in addition to PDO, for example, also MySQLi and the older version MySQL (the extension, not the DBMS). Thus, when one uses `Zend\Db\Adapter` instead of the native functions and objects, one increases portability. If we again return again to the definitions given above, we could designate `Zend\Db\Adapter` as our "data access object".

In order for `Zend\Db` to function properly, the database connection must initially be made via the `Adapter`:

```php
<?php
$adapter = new \Zend\Db\Adapter\Adapter(
        array(
                'driver' => 'Pdo_Mysql',
                'hostname' => 'localhost'
                'database' => 'app',
                'username' => 'root',
                'password' => ''
        )
);
```

Listing 14.1

Using passwords

In a production system, we naturally ensure that we use strong passwords and even better do not work with the user "root" at all, but with special users whose options can be restricted to the most essential ones.

The MySQL service was made available beforehand on the `localhost`, the database `app` was setup and the user `root` was made accessible without password (this is certainly okay in an development system). In addition, the `log` table was set up with the aid of the following SQL statement:

```sql
CREATE TABLE log (
        id int(10) NOT NULL auto_increment,
        ip varchar(16) NOT NULL,
        timestamp varchar(10) NOT NULL,
        PRIMARY KEY (id)
);
```

Generating and running SQL-Statements

If one has added a few data sets to the table manually, they can be accessed via the previously generated `Adapter`.

```php
<?php
$stmt = $adapter->createStatement('SELECT * FROM log');
$results = $stmt->execute();

foreach($results as $result)
var_dump($result);
```

Listing 14.2

A very elegant, object-oriented procedure for generating SQL statements is `Zend\Db\Sql`:

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

Listing 14.3

It is also possible to easily add a `WHERE` clause:

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

Listing 14.4

The same applies for the other customary constructives—such as `LIMIT`, `OFFSET`, `ORDER`, etc.—whose usage is fundamentally self-explanatory. Adding a `JOIN` is also essentially simple:

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

Listing 14.5

In this case, the additional `host` table is referenced; indeed, the latter is generated with the following statement:

```sql
CREATE TABLE host (
        id int(10) NOT NULL auto_increment,
        ip varchar(16) NOT NULL,
        hostname varchar(100) NOT NULL,
        PRIMARY KEY (id)
);
```

This “join” is an “inner join”. Other types of “joins” are supported in a similar manner:

```php
<?php
$sql = new \Zend\Db\Sql\Sql($adapter);
$select = $sql->select();
$select->from('log');

$select->join('host',
   'host.ip = log.ip',
   array('*'),
   \Zend\Db\Sql\Select::JOIN_LEFT
);

$select->where(array('log.ip' => '127.0.0.1'));
$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
```

Listing 14.6

In this case, the `array('*')` parameter indicates the columns which are to be transferred to the result from the `host` table in the scope of the “join”. When `array('*')` is used, all columns are utilised; under specification of specific columns, the result can be restricted; and an associative array can be employed to utilise alias values for the columns under consideration.

In this manner, data can also be written elegantly into the database

```php
<?php
$sql = new \Zend\Db\Sql\Sql($adapter);
$insert = $sql->insert('host');
$insert->columns(array('ip', 'hostname'));
$insert->values(array('192.168.1.15', 'michaels-ipad'));
$statement = $sql->prepareStatementForSqlObject($insert);
$results = $statement->execute();
```

Listing 14.7

updated

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

Listing 14.8

and deleted:

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

Listing 14.9

The multifarious options

Additional details about `Zend\Db\Sql\Sql` can also be found in the official documentation[a] as needed.

[a] http://packages.zendframework.com/docs/latest/manual/en/modules/zend.db.sql.html

Working with tables and entries

Using `Zend\Db\TableGateway` makes things much simpler. As discussed in the introduction to this chapter, a `TableGateway` object represents a table in a database. A query can be realised as follows:

```php
<?php
$hostTable = new \Zend\Db\TableGateway\TableGateway('host', $adapter);
$results = $hostTable->select(array('hostname' => 'michaels-mac'));

foreach ($results as $result)
var_dump($result);
```

Listing 14.10

Also in this case, the appropriate adapter was generated beforehand:

```php
<?php
$adapter = new \Zend\Db\Adapter\Adapter(
        array(
                'driver' => 'Pdo_Mysql',
                'database' => 'app',
                'username' => 'root',
                'password' => ''
)
);
```

Listing 14.11

If one desires to further process the returned data, for example revise or delete data sets, “row data gateway” objects can be requested; they represent a data set in the respective table and provide functions to manipulate them. To do this, the requests given above must be modified as follows:

```php
<?php
$hostTable = new \Zend\Db\TableGateway\TableGateway(
        'host',
        $adapter,
        new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
);

$results = $hostTable->select(array('hostname' => 'michaels-mac'));

foreach ($results as $result)
var_dump($result);
```

Listing 14.12

In generating the `TableGateway`, we transfer the `RowGatewayFeature` as additional parameter. As a result of this, the `TableGateway` no longer makes the individual results of the query available as arrays or `ArrayObjects` , but instead as a collection of `RowGateway` objects. The latter provide a `save()` und `delete()` method that allows one to return alterations in the data set to the database or even to delete the former:

```php
<?php
$hostTable = new \Zend\Db\TableGateway\TableGateway(
        'host',
        $adapter,
        new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
);

$results = $hostTable->select(array('ip' => '127.0.0.1'));
$result = $results->current();
$result->hostname = 'michaels-macbook';
$result->save(); // oder: $result->delete();
```

Listing 14.13

The `Row Data Gateway` objects that are returned by a request are all of the `Zend\Db\RowGateway\RowGateway` type. They thus provide the added value that they can ensure their own persistence, but, on the other hand, they do not carry any specialist information. In this case, the so-called “hydration”, in which the data loaded from the database are transferred transparently into a specialist object, comes into play In the process, the object indeed looses its persistence functions, but in reading operations one can initially easily dispense with that.

To begin with, we create the specialist object, a so-called “entity”, in the `Helloworld` module under `/src/Helloworld/Entity/Host.php`:

```php
<?php
namespace Helloworld\Entity;

class Host
{
        protected $ip;
        protected $hostname;

        public function getHostname()
        {
                return $this->hostname;
        }

        public function getIp()
        {
                return $this->ip;
        }
}
```

Listing 14.14

Initially, the `Host` has only two `protected` characteristics and “getter”. The following request now results in objects of the `host` type being maintained in the `host` variable:

```php
<?php
$hostTable = new \Zend\Db\TableGateway\TableGateway(
'host',
$adapter,
new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
);

$results = $hostTable->select(array('ip' => '127.0.0.1'));

$hosts = new \Zend\Db\ResultSet\HydratingResultSet(
        new \Zend\Stdlib\Hydrator\Reflection(),
        new \Helloworld\Entity\Host()
);

$hosts->initialize($results->toArray());
```

Listing 14.15

The result of the query is transferred into a `HydratingResultSet`. To achieve this, a decision on how the data should be assigned (`Reflection`) and where they should ultimately end up (`Host`) must be made. To achieve this, the so-called “prototype pattern” is used, in which an exemplary object (in this case the newly generated instance of the `Host` class) duplicates the query (`clone`) for every dataset in the `ResultSet` and equips them with the data. Thus, one inserts a “prototypical” object and obtains as many clones of this object (each initialised with the correct data) as required.

But why aren’t the required objects simply instantiated via the `new` operator as required? The idea behind this is that the object (in addition to the data fields that are to be filled) can also have additional dependencies to further objects that could not be resolved automatically and without problems. Instead, an already configured object is simply cloned and the problem, thus avoided.

The assumption which is made in this case is that the names of the table columns agree with the object characteristics. If this is not (always) the case, one thus re-acquires only partially filled objects in some cases.

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

The `hostname` is empty because the corresponding database field is designated as `workstation`. Now, one can either rename the database field "hostname" or the object characteristic "workstation". However, since this is not always appropriate or possible, one can instead make do with a derived "hydrator" of one's own, which takes care of the required "mapping":

```php
<?php
namespace Helloworld\Mapper;

use Zend\Stdlib\Hydrator\Reflection;
use Helloworld\Entity\Host;

class HostHydrator extends Reflection
{
        public function hydrate(array $data, $object)
        {
                if (!$object instanceof Host) {
                        throw new \InvalidArgumentException(
                                '$object must be an instance of Helloworld\Entity\Host'
                        );
                }

                $data = $this->mapField('workstation', 'hostname', $data);
                return parent::hydrate($data, $object);
        }

        protected function mapField($keyFrom, $keyTo, array $array)
        {
                $array[$keyTo] = $array[$keyFrom];
                unset($array[$keyFrom]);
                return $array;
        }
}
```

Listing 14.16

This class is located in the `src/Helloworld/Mapper/HostHydrator.php` directory and accordingly maps the `workstation` field onto the `hostname` field. Now, one must only modify the invocation such that the appropriate "hydrator" is applicable:

```php
<?php
$hostTable = new \Zend\Db\TableGateway\TableGateway(
        'host',
        $adapter,
        new \Zend\Db\TableGateway\Feature\RowGatewayFeature('id')
);

$results = $hostTable->select(array('ip' => '127.0.0.1'));

$hosts = new \Zend\Db\ResultSet\HydratingResultSet(
        new \Helloworld\Mapper\HostHydrator(),
        new \Helloworld\Entity\Host()
);

$hosts->initialize($results->toArray());
var_dump($hosts->current());
```

Listing 14.17

Now, the loading of the data again functions as desired:

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

Organisation of database queries

Until now we have made or executed the database connection and the database queries directly in a controller. For demonstration purposes that was certainly acceptable. However, in an actual application one requires a better procedure for dealing with database queries. The following procedure is recommended: The creation of a database adapter, which allows access to the database, is shifted into the `ServiceManager`, the connection data deposited in a configuration file, and the data queries encapsulated around the individual tables or entities, respectively, in special objects.

Creating a database adapter via the ServiceManager

In order to prepare the `ServiceManager` for the creation of a database adapter, we initially modify the `module.config.php` of the `Helloworld` module as follows:

```php
<?php
// [..]
'service_manager' => array(
    'factories' => array(
        'Zend\Db\Adapter\Adapter' => function ($sm) {
            $config = $sm->get('Config');
            $dbParams = $config['dbParams'];

            return new Zend\Db\Adapter\Adapter(array(
                'driver'    => 'pdo',
                'dsn'                =>
                                    'mysql:dbname='.$dbParams['database']
                                            .';host='.$dbParams['hostname'],
                'database'  => $dbParams['database'],
                'username'  => $dbParams['username'],
                'password'  => $dbParams['password'],
                'hostname'  => $dbParams['hostname'],
            ));
        },
    ),
)
// [..]
```

Listing 14.18

Any already existing service definitions should naturally be retained as required. Incidentally, the module in which this service definition is located is absolutely arbitrary. As shown above in the `HelloWorld` module, we can also accommodate it in `Application` in some cases. If one desires to use the database adapter across a number of "function modules", it is appropriate to move the definition into the `Application` module, simply because one knows "by convention" where one must look when one is looking for the definition of a service that is used across several modules.

The above-mentioned call back function creates the database adapter and to achieve this accesses connection data that are deposited in a configuration file. I used the `dev1.local.php` file for this purpose:

```php
<?php
return array(
        'dbParams' => array(
                'database'  => 'app',
                'username'  => 'root',
                'password'  => '',
                'hostname'  => 'localhost',
        )
);
```

Listing 14.19

Thus, we can charge the `ServiceManager` with the creation of the database adapter where ever it is required: Out of

```php
<?php
$adapter = new \Zend\Db\Adapter\Adapter(
        array(
                'driver' => 'Pdo_Mysql',
                'database' => 'app',
                'username' => 'root',
                'password' => ''
        )
);
```

Listing 14.20

Located, for example, in a controller, therefore becomes

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

Listing 14.21

Capsules of similar queries

To avoid scattering database queries across the entire application and to prevent alterations of data schemes or even the query itself from becoming maintenance nightmares, it is advisable to consolidate all queries which refer to the same “entity” or database table, respectively, at one location from the very beginning. To achieve this, one can effectively use the `TableGateway`, which we are already familiar with and which allows simple access to database tables. We set up a specific `TableGateway` for every entity. In this context, we orient ourselves to the respective entity:

```php
<?php
namespace Helloworld\Mapper;

use Helloworld\Entity\Host as HostEntity;
use Zend\Stdlib\Hydrator\HydratorInterface;
use Zend\Db\TableGateway\TableGateway;
use Zend\Db\TableGateway\Feature\RowGatewayFeature;

class Host extends TableGateway
{
        protected $tableName  = 'host';
        protected $idCol = 'id';
        protected $entityPrototype = null;
        protected $hydrator = null;

        public function __construct($adapter)
        {
                parent::__construct($this->tableName,
                        $adapter,
                        new RowGatewayFeature($this->idCol)
                );

                $this->entityPrototype = new HostEntity();
                $this->hydrator = new HostHydrator();
        }

        public function findByIp($ip)
        {
                return $this->hydrate(
                        $this->select(array('ip' => $ip))
                );
        }

        public function hydrate($results)
        {
                $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
                    $this->hydrator,
                        $this->entityPrototype
                );

                return $hosts->initialize($results->toArray());
        }
}
```

Listing 14.22

We deposit the code for the mapper in `src/Helloworld/Mapper/Host.php` and thus in the same directory in which our `HostHydrator` is also already located and which we also again make use of in this case.

Naturally, this code can be further optimised; there is no question about that. For example, it would be more appropriate to inject all the configuration values and dependencies that to hardcode them there. But for this example it is sufficient to do it simply. When one desires to make it even easier for oneself, one draws on the ready-to-use AbstractDbMapper[58] of the ZF-Commons[59] git repositories. This work has already been done, quasi by the "head office" (the repository is maintained by developers who are also directly involved in Framework themselves). It's very worthwhile to take a look at it in any case.

The invocation still required in the controller is already very compact:

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

Listing 14.23

We could shorten the code even more if we were to inject the `Adapter` automatically at the creation of the "mapper":

```php
<?php
// [..]
'service_manager' => array(
    'factories' => array(
        'Zend\Db\Adapter\Adapter' => function ($sm) {
            $config = $sm->get('Config');
            $dbParams = $config['dbParams'];

            return new Zend\Db\Adapter\Adapter(array(
                'driver'    => 'pdo',
                'dsn'       =>
                                       'mysql:dbname='.$dbParams['database']
                                       .';host='.$dbParams['hostname'],
                'database'  => $dbParams['database'],
```

[58]https://github.com/ZF-Commons/ZfcBase/blob/master/src/ZfcBase/Mapper/AbstractDbMapper.php
[59]https://github.com/ZF-Commons

```
                'username'  => $dbParams['username'],
                'password'  => $dbParams['password'],
                'hostname'  => $dbParams['hostname'],
            ));
        },
        'Helloworld\Mapper\Host' => function ($sm) {
            return new \Helloworld\Mapper\Host(
                $sm->get('Zend\Db\Adapter\Adapter')
            );
        }
    ),
),
// [..]
```

Listing 14.24

Now, the invocation still required in the controller is just a one-liner:

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

Listing 14.25

By utilising the `TableGateway`, we can now also elegantly solve the problem of the entities loosing their persistence functions as a result of hydration. However, an alteration of an entity can now no longer be directly written into the database by using `save()`. We now instruct the `Host` mapper to do this:

```
<?php

namespace Helloworld\Mapper;

use Helloworld\Entity\Host as HostEntity;
use Zend\Stdlib\Hydrator\HydratorInterface;
use Zend\Db\TableGateway\TableGateway;
use Zend\Db\TableGateway\Feature\RowGatewayFeature;
use Zend\Db\Sql\Sql;
use Zend\Db\Sql\Insert;

class Host extends TableGateway
{
        protected $tableName  = 'host';
```

```
    protected $idCol = 'id';
    protected $entityPrototype = null;
    protected $hydrator = null;

    public function __construct($adapter)
    {
        parent::__construct($this->tableName,
            $adapter,
            new RowGatewayFeature($this->idCol)
        );

        $this->entityPrototype = new HostEntity();
        $this->hydrator = new HostHydrator();
    }

    public function findByIp($ip)
    {
        return $this->hydrate(
            $this->select(array('ip' => $ip))
        );
    }

    public function hydrate($results)
    {
        $hosts = new \Zend\Db\ResultSet\HydratingResultSet(
            $this->hydrator,
            $this->entityPrototype
        );

        return $hosts->initialize($results->toArray());
    }

    public function insert($entity)
    {
        return parent::insert($this->hydrator->extract($entity));
    }

    public function updateEntity($entity)
    {
        return parent::update(
            $this->hydrator->extract($entity),
            $this->idCol . "=" . $entity->getId()
```

```
                );
        }
}
```

Listing 14.26

The `insert()` und `updateEntity()` methods are new here. However, we must also extend the `HostHydrator` if the column designations deviate from the object characteristics:

```php
<?php
namespace Helloworld\Mapper;

use Zend\Stdlib\Hydrator\Reflection;
use Helloworld\Entity\Host as HostEntity;

class HostHydrator extends Reflection
{
        public function hydrate(array $data, $object)
        {
                if (!$object instanceof HostEntity) {
                        throw new \InvalidArgumentException(
                                '$object must be an instance of Helloworld\Entity\Host'
                        );
                }

                $data = $this->mapField('workstation', 'hostname', $data);
                return parent::hydrate($data, $object);
        }

        public function extract($object)
        {
                if (!$object instanceof HostEntity) {
                        throw new \InvalidArgumentException(
                                '$object must be an instance of Helloworld\Entity\Host'
                        );
                }

                $data = parent::extract($object);
                $data = $this->mapField('hostname', 'workstation', $data);
                return $data;
        }

        protected function mapField($keyFrom, $keyTo, array $array)
        {
                $array[$keyTo] = $array[$keyFrom];
                unset($array[$keyFrom]);
                return $array;
        }
}
```

Listing 14.27

We can now alter a previously loaded dataset in the controller:

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

$host = $hosts->current();
$host->setHostname('my-mac');

$this->getServiceLocator()
        ->get('Helloworld\Mapper\Host')->updateEntity($host);
```

Listing 14.28

The insertion of new datasets by the `Host`-Mapper now functions similarly:

```
<?php
$newEntity = new \Helloworld\Entity\Host();
$newEntity->setHostname('michaels-iphone');
$newEntity->setIp('192.168.1.56');

$this->getServiceLocator()
        ->get('Helloworld\Mapper\Host')->insert($newEntity);
```

Listing 14.29

For completeness sake, here is the `Host`-Entity in its final configuration level again:

```php
<?php
namespace Helloworld\Entity;

class Host
{
	protected $id;
	protected $ip;
	protected $hostname;

	public function getHostname()
	{
		return $this->hostname;
	}

	public function getIp()
	{
		return $this->ip;
	}

	public function setIp($ip)
	{
		$this->ip = $ip;
	}

	public function setHostname($hostname)
	{
		$this->hostname = $hostname;
	}

	public function setId($id)
	{
		$this->id = $id;
	}

	public function getId()
	{
		return $this->id;
	}
}
```

Listing 14.30

In particular, the `id` characteristic is also still important here because the dataset that is to be updated is determined in this manner.

Zend\DB alternative: Doctrine 2 ORM

If we are completely honest: `Zend\Db` is indeed useful but it quickly reaches its limits. The challenge of “persistence” in complex systems cannot be satisfactorily mastered in more complex systems by using it. `Zend\Db` does not provide either an “active record” implementation or the functionality of an “ORM”; which, however, sooner or later very noticeably contributes to keeping the complexity of an application manageable. In addition, one has to write a large amount of code dealing with “persistence” oneself. I therefore absolutely recommend that you take a look at Doctrine 2[60]. Zend Framework 2 and Doctrine 2 complement each other superbly and really only develop their strengths completely when used together.

[60]http://www.doctrine-project.org/

Validators

Standard validators

With `Zend\Validator`, Framework provides a simple, but very helpful mechanism for validating values with regard to defined requirements, for example entries sent in the scope of a POST request by the client For the most common requirements, it also additionally provides specific implementations. For example, with just a few lines of code, a value can be checked for conformity with the ISBN Standard:

```php
<?php
$validator = new \Zend\Validator\Isbn();

if($validator->isValid('315000017'))
        echo "In Ordnung!";
```

Listing 15.1

The fact that this is a syntactic and not a semantic check is important. Thus, "O.K." is also displayed if an ISBN is specified that is indeed correct, but has not yet been assigned to any book.

Validators generally provide error messages for extremely different error situations, which can be invoked subsequent to a validation procedure by means of `getMessages()`:

```php
<?php
$validator = new \Zend\Validator\Isbn();

if($validator->isValid('315090017'))
        echo "In Ordnung!";
else {
        foreach ($validator->getMessages() as $messageId => $message) {
                echo $message;
        }
}
```

Listing 15.2

In this case, we see

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

on the screen. However, if we invoke,

```php
<?php
$validator = new \Zend\Validator\Isbn();

if($validator->isValid('12.23'))
        echo "In Ordnung!";
else {
        foreach ($validator->getMessages() as $messageId => $message) {
                echo $message;
        }
}
```

Listing 15.3

this results in output of the following:

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

The `Isbn` validator thus two encompasses two different error cases. As can already be expected, one can set individual error messages by means of the `setMessage()` method:

```php
<?php
$validator = new \Zend\Validator\Isbn();

$validator->setMessage(
        'Es wird ein String oder ein Integer-Wert zur Validierung benötigt!',
        \Zend\Validator\Isbn::INVALID
);

if($validator->isValid(12.23))
        echo "In Ordnung!";
else {
        foreach ($validator->getMessages() as $messageId => $message) {
                echo $message;
        }
}
```

Listing 15.4

To achieve this, the message key that is used in the validator is to be specified in each case.

The following self-explanatory standard validators already exist in Framework:

- Barcode

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

Performing several validations simultaneously

With the aid of the `Zend\Validator\ValidatorChain` class, several validators can be linked for sequential checks, as required.

Writing your own validators

If one requires additional validation functions, one can easily write one's own validators if they inherit from `AbstractValidator`:

```php
<?php
class Helloworld\Validator\Float extends Zend\Validator\AbstractValidator
{
        const FLOAT = 'float';

        protected $messageTemplates = array(
                self::FLOAT => "'%value%' ist kein Float-Wert."
        );

        public function isValid($value)
        {
                $this->setValue($value);

                if (!is_float($value)) {
                        $this->error(self::FLOAT);
                        return false;
                }

                return true;
        }
}
```

Listing 15.5

Webforms

Webforms are integral components of web applications: basically they are the only possibility that a user has to transfer data to the server, i.e. to the application. Webforms always primarily present the application developer with many-faceted challenges. Comprehensive support for webforms are an integral part of a good web-framework, and thus exorcises our fear of them to some extent. With `Zend\Form`, Zend Framework 2 provides a high-performance solution, which can do nearly everything that one desires. Even if it is not particularly easy to understand.

To begin with we need a bit of theory: Each webform is fundamentally based on one or more objects of the `Zend\Form\Element` type. They form the basic unit of `Zend\Form`-based forms. In this context, one element initially corresponds fundamentally to the customary HTML form elements, which one is already familiar with, i.e. input fields (`text`), radio buttons (`radio`), selection lists (`select`) and so on The filter that should be employed for the respectively input data and the rules that are to be used for the validation of that data can be specified via a so-called "input", as soon as the former have been received. Typical filters are, for example, `StripTags`, which remove all HTML tags in the entered character strings or `StringToLower`, which (as already expected) converts all input characters to small letters. Validation means the examination of the input data for compliance with previously defined conditions. Thus, for example, the ISBN Validator can be used for an input field to check whether the data correspond to an ISBN number, whereas `NotEmpty` checks whether at least something has been entered into the field. The individual "inputs" are aggregated in the so-called "InputFilter" and the respective webform is made available. The individual form elements can, in turn, be semantically grouped as so-called "fieldsets". Incidentally, from a technical point of view, a `Zend\Form\Fieldset` is also again only an "Element" in this context, derived from the `Zend\Form\Element` class. "Fleldsets" or individual elements (or also both together) are then, in turn, combined to form a `Zend\Form\Form`, which is technically again only an "Element" —it, too, is derived from `Zend\Form\Element`.

Preparing a form

A webform is quickly prepared with HTML:

```html
<form action="#" method="post">
<fieldset>
<label for="name">Ihr Name:</label>
<input type="text" id="name" />
<label for="email">Ihre E-Mail-Adresse:</label>
<input type="email" id="email" />
<input type="submit" value="Eintragen" />
</fieldset>
</form>
```

Listing 16.1

However, the experienced application developer knows that this does not mean that everything has been achieved—not by a long shot: The received data has to be syntactically and sematically validated for further processing—for example, for storage in a database—and error situations, treated with proven means, not least because checks of the input data by the client, even just for safety reasons, are never adequate. All this code must normally be developed manually and the corresponding code realised, for example, in the controller. Framework can help in this case. The fundamental idea is to realise a webform as a distinct construct that contains all the information on the contained fields and their basic syntactic and semantic conditions as well as on the necessary information with regard to how the data enter and subsequently exit via the best pathways. In Framework, a number of different classes and components are used for webforms.

- `Zend\Form`: core components via which the application developers interact with webforms.
- `Zend\InputFilter`: enhances webforms with filter and validation capabilities.
- `Zend\View\Helper`: allows the visual presentation of forms.
- `Zend\Stdlib\Hydrator`: enhances webforms with the capability to automatically transfer data from the form into other objects or to import it from there.

Framework allows the application developer a number of decisions as to how a webform is to be structured. In this context, the pendulum oscillates—as is often the case in Zend Framework 2—between writing code and writing configuration. Other than that, there is a wide range of options for dealing with webforms. A recommendable approach is to map forms via one's own `Form` classes. If we were to desire to depict the form shown above for the regestration to a newsletter via a `Zend\Form` Object, we have to define the webform in the `src/Helloworld/Form/SignUp.php` file.

```php
<?php
namespace Helloworld\Form;

use Zend\Form\Form;

class SignUp extends Form
{
        public function __construct()
        {
                parent::__construct('signUp');
                $this->setAttribute('action', '/signup');
                $this->setAttribute('method', 'post');

                $this->add(array(
                        'name' => 'name',
                        'attributes' => array(
                                'type'  => 'text',
                                'id' => 'name'
                        ),
                        'options' => array(
                                'label' => 'Ihr Name:'
                        ),
                ));

                $this->add(array(
                        'name' => 'email',
                        'attributes' => array(
                                'type'  => 'email',
                                'id' => 'email'
                        ),
                        'options' => array(
                                'label' => 'Ihre E-Mail-Adresse:'
                        ),
                ));

                $this->add(array(
                        'name' => 'submit',
                        'attributes' => array(
                                'type'  => 'submit',
                                'value' => 'Eintragen'
                        ),
                ));
        }
}
```

Listing 16.2[61]

New elements or fieldsets are added to the form via the `add()` method. To achieve this, the `add()` method, in turn, invokes the `Zend\Form\Factory`; the latter knows how to interpret the specification (termed "Spec"), which is transferred as an array, and to generate the appropriate elements. The most important components of the specification are `name`, `type`, `attributes` und `options`. The `name` can be freely selected, whereas the `type` represents an element type, which really exists in Framework (in this case elements of the `Zend\Form\Element\Text` type are generated because nothing else has been specified) and `attributes` the establishment of all attributes of the ultimately generated HTML element (`<input name="name" type="text" id="name">`). The `options` section allows the definition of field labels (i.e. that which is additionally displayed before or above the input field proper) via `label` or, on the other hand, also the attribute of the label via `label_attributes`, respectively.

[61]https://gist.github.com/3919928

In addition to the individual elements, a number of attributes will be stipulated by the application developers via `setAttribute()`, among them `action` and `method`, which are both absolutely necessary so that the webform can also be subsequently sent off.

Displaying a form

I have good news for all those who have used Version 1 of Framework: There are no longer any form decorators: Form-decorators were a very committed, but at the same time also very complicated, approach, which allowed the rendering of webforms by means of nesting of view objects, each of which then generated a small part of the final HTML markup. I believe that Form-Decorators war just about the most difficult aspect of Zend Framework 1 and also generated a large number of problems otherwise, which one had not even thought of beforehand. But, that's enough reminiscing; they no longer exist in Version 2. The advantage is very clear: the simpler use of forms, but the code is slightly less compact if one just uses the means available in the programme. To display the form defined above, somewhat more view code is required as for the `<?php echo $this->form; ?>` of former times. However, this can also be realised with a little bit of effort, as we will see in the practice part of the book.

```php
<?php
$this->form->prepare();
echo $this->form()->openTag($this->form);
echo $this->formRow($this->form->get('name'));
echo $this->formRow($this->form->get('email'));
echo $this->formSubmit($this->form->get('submit'));
echo $this->form()->closeTag();
```

Listing 16.3

This code has to be in the `View` file of the `Action` in which the Form is instantiated and will be returned in the scope of the `ViewModel`:

```php
<?php
return new ViewModel(
        array(
                'form' => new \Helloworld\Form\SignUp()
        )
);
```

Listing 16.4

A number of view helpers and the form itself are used for the depiction. Roughly stated, there is an appropriate `ViewHelper`, which can be used for the depiction, for every type of form element that HTML defines. Moreover, there are a number of additional auxiliary constructs, such as `FormRow`, which ensures that a form field with its label, the field itself and, as required, any existing error message are displayed analogously "in a sequence". The `View Helpers` are all nearly self-explanatory. It is important that `prepare()` is invoked to begin with, before any other elements are accessed; otherwise, this results in an error.

Editing form entries

Basically, there are two possibilities of realising form processing: either in the same action in which the empty webform was generated or in a separate action. If the processing is to take place in the same action, an "isPost() check" can be used to determine whether the form is to be displayed or whether the entries are to be edited:

```php
<?php
if ($this->getRequest()->isPost()) {
        // Formularverarbeitung
} else {
        return new ViewModel(
                array(
                        'form' => new \Helloworld\Form\SignUp()
                )
        );
}
```

Listing 16.5

To access the sent data, one can now either directly access the POST data:

```php
<?php
$form = new \Helloworld\Form\SignUp();

if ($this->getRequest()->isPost()) {
        $data = $this->getRequest()->getPost();
        var_dump($data);exit;
} else {
        return new ViewModel(
                array(
                        'form' => $form
                )
        );
}
```

Listing 16.6

in which the following output is generated:

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

or one can access the sent data via the `Form` object. However, that is only then possible if one has previously validated the data via the web form.

Validating form entries

Up to now, the advantage of `Zend\Form` is admittedly relatively straightforward, but now we're going to take off. If we do not obtain the sent data via the POST array of PHP, but rather via the form itself, the entries can be automatically validated in a simple manner based on previously defined rules and the data filtered.

To achieve this, the so-called "InputFilter" has to be defined initially. The term "InputFilter" is a bit misleading because not only the filters but also the validators are defined in this way. Filters modify the input data as required, whereas validators test the data for certain conditions, for example a maximum string length.

There are a number of options for the definition of the "InputFilter". For example, the definitions can be moved into a class of their own or alternatively they are made available by the form class via the `getInputFilter()` method. In this case, we realize the "InputFilter" in a class of its own in the `src/Helloworld/Form/SignUpFilter.php` file:

```
<?php
namespace Helloworld\Form;

use Zend\Form\Form;
use Zend\InputFilter\InputFilter;

class SignUpFilter extends InputFilter
{
        public function __construct()
        {
                $this->add(array(
                        'name' => 'email',
                        'required'=> true,
                        'validators' => array(
                                array(
                                        'name' => 'EmailAddress'
                                )
                        ),
                ));

                $this->add(array(
                        'name' => 'name',
                        'required' => true,
                        'filters' => array(
                                array(
                                        'name' => 'StringTrim'
                                )
                        )
                ));
        }
}
```

Listing 16.7

In this case, the individual inputs are added to the “InputFilter” by the `add()` method. In this context, an input basically corresponds to a form element, i.e. an input option, which is reference via the `name` and is subsequently taken into account. Whether or not the field is mandatory can be controlled by means of `required`. Analogously, the filters and validators to be used can be defined using `validators` and `filters`, where the `name` must correspond to the filters or validators supplied with Framework or to subsequently added filters or validators, respectively. A look at the source code of the `Zend\Validator\ValidatorPluginManager` reveals the standard validators and their symbolic names, via which they can be accessed/invoked. `Zend\Filter\FilterPluginManager` provides information on Framework’s filters.

Now we must first extend the form definition with regard to the `SignUpFilter`.

```
<?php
namespace Helloworld\Form;

use Zend\Form\Form;

class SignUp extends Form
{
        public function __construct()
        {
                parent::__construct('signUp');
                $this->setAttribute('action', '/signup');
                $this->setAttribute('method', 'post');
                $this->setInputFilter(new \Helloworld\Form\SignUpFilter());

                $this->add(array(
                        'name' => 'name',
                        'attributes' => array(
                                'type'  => 'text',
                        ),
                        'options' => array(
                                'id' => 'name',
                                'label' => 'Ihr Name:'
                        ),
                ));

                $this->add(array(
                        'name' => 'email',
                        'attributes' => array(
                                'type'  => 'email',
                        ),
                        'options' => array(
                                'id' => 'email',
                                'label' => 'Ihre E-Mail-Adresse:'
                        ),
                ));

                $this->add(array(
                        'name' => 'submit',
                        'attributes' => array(
                                'type'  => 'submit',
                                'value' => 'Eintragen'
                        ),
                ));
        }
}
```

Listing 16.8[62]

The `setInputFilter()` method links the form with the "InputFilter" in this case. But also in this case, the general advice that it is advisable not the hard wire the `SignUpFilter` applies; instead it should be injected via an appropriate factory and the `ServiceManager` or alternatively via `Zend\Di`.

Now, form processing can be effected in the controller as follows:

[62]https://gist.github.com/3920184

```php
<?php
public function indexAction()
{
        $form = new \Helloworld\Form\SignUp();

        if ($this->getRequest()->isPost()) {
                $form->setData($this->getRequest()->getPost());

                if ($form->isValid()) {
                        var_dump($form->getData());
                } else {
                        return new ViewModel(
                                array(
                                        'form' => $form
                                )
                        );
                }
        } else {
                return new ViewModel(
                        array(
                                'form' => $form
                        )
                );
        }
}
```

Listing 16.9[63]

In this case, a successful input results in the pleasant output:

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

Thus, the data are now present in the form of an array for further processing. Any filters which were registered for the respective fields have already been applied by this time.

Incidentally, in this manner, error messages are now also already displayed; if this is the case, the form will not be validated, and we return it to the view for repeated display. If one should happen to send the form without entries (which should result in an error according to the `SignUpFilter` definition), one sees the following for the two fields:

[63] https://gist.github.com/3920196

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

In order to change the displayed error message, one uses the `options` array in the validator definition. However, the error message shown above has a special feature, because the `SignUpFilter` does not yet even use the `NotEmpty` validator, which we were able to provide with the necessary "options". Instead, the requirement that the field must be filled in is expressed via `'required' => true`. Based on this, Framework automatically generates the appropriate validator. Thus, the `'required' => true` expression is a "shortcut". When we adapt the filter as follows, we can deposit individual error messages:

```php
<?php
namespace Helloworld\Form;

use Zend\Form\Form;
use Zend\InputFilter\InputFilter;

class SignUpFilter extends InputFilter
{
        public function __construct()
        {
                $this->add(array(
                        'name' => 'email',
                        'validators' => array(
                                array(
                                        'name' => 'NotEmpty',
                                        'options' => array(
                                                'messages' => array(
                                \Zend\Validator\NotEmpty::IS_EMPTY =>
                                'Bitte geben Sie etwas ein.'
                                                )
                                        )
                                ),
                                array(
                                        'name' => 'EmailAddress',
                                        'options' => array(
                                                'messages' => array(
                                \Zend\Validator\EmailAddress::INVALID_FORMAT =>
                                'Bitte richtige E-Mail-Adresse eingeben.'
                                                )
                                        )
                                ),
                        ),
                ));

                $this->add(array(
                        'name' => 'name',
                        'filters' => array(
                                array(
                                        'name' => 'StringTrim'
                                )
                        ),
                        'validators' => array(
                                array(
                                        'name' => 'NotEmpty',
                                        'options' => array(
                                                'messages' => array(
                                \Zend\Validator\NotEmpty::IS_EMPTY =>
                                'Bitte geben Sie etwas ein.'
                                                )
                                        )
                                )
                        )
                ));
        }
}
```

Listing 16.10

If we now send the form without entering an email address, we become the following error message:

```
Please input something.
Please enter the correct email address.
```

Or to be more exact: We see 2 error messages at the same time. One should know that in the course of processing the validators are processed one after another and any occurring error messages are collected. Then all the error messages are displayed when the appropriate “View Helper” is employed. In most cases, however, that is not really helpful for the user; instead, a single error message would be adequate. To ensure this, the `'break_chain_on_failure' => true` option can be set. It insures that, after a validator failure, the subsequent ones are not implemented at all and no possible additional error messages are generated:

```php
<?php
namespace Helloworld\Form;

use Zend\Form\Form;
use Zend\InputFilter\InputFilter;

class SignUpFilter extends InputFilter
{
        public function __construct()
        {
                $this->add(array(
                        'name' => 'email',
                        'validators' => array(
                                array(
                                        'name' => 'NotEmpty',
                                        'break_chain_on_failure' => true,
                                        'options' => array(
                                                'messages' => array(
                                \Zend\Validator\NotEmpty::IS_EMPTY =>
                                'Bitte geben Sie etwas ein.'
                                                )
                                        )
                                ),
                                array(
                                        'name' => 'EmailAddress',
                                        'options' => array(
                                                'messages' => array(
                                \Zend\Validator\EmailAddress::INVALID_FORMAT =>
                                'Bitte richtige E-Mail-Adresse eingeben.'
                                                )
                                        )
                                ),
                        ),
                ));

                $this->add(array(
                        'name' => 'name',
                        'filters' => array(
                                array(
                                        'name' => 'StringTrim'
                                )
                        ),
                        'validators' => array(
                                array(
                                        'name' => 'NotEmpty',
                                        'options' => array(
                                                'messages' => array(
                                \Zend\Validator\NotEmpty::IS_EMPTY =>
                                'Bitte geben Sie etwas ein.'
                                                )
                                        )
                                )
                        )
                ));
        }
}
```

Listing 16.11[64]

Since the individual validators can generate different error messages depending on the situation, the message must be deposited at the appropriate key, which is accessible via a constant in the respective class. If you have any doubts, a look at the code of the respective validator also helps.

Standard form elements

As shown in the previous examples, one can generate any arbitrary HTML elements by choosing the appropriate filters and validators. However, depending on the element, this procedure can require a lot of effort, particularly because all the configurations must be made manually. For this situation, Framework has a number of options still open: a large number of "preconfigured" elements, which immediately provide the necessary configuration.

- Button
- Captcha
- Checkbox
- Collection
- Color
- Csrf
- Date
- DateTime
- DateTimeLocal
- Email

[64]https://gist.github.com/3920507

- File
- Hidden
- Image
- Month
- MultiCheckbox
- Number
- Password
- Radio
- Range
- Select
- Submit
- Text
- Textarea
- Time
- URL
- Week

Take for example the `Number` element. We would like to generate an input field in which the numerical values of a defined range may be entered. If we were to set up these rules ourselves, we would have to configure a large number of validators, among them:

- `NumberValidator`: only numbers can be entered.
- `GreaterThanValidator`: the entered value must be more than or equal to the minimum.
- `LessThanValidator`: the entered value must be less than or equal to the maximum.
- `StepValidator`: only whole-numbered values are accepted.

We can dispense with this work when we use the `Zend\Form\Element\Number` directly:

```php
<?php
// [..]
$this->add(array(
        'name' => 'age',
        'type' => 'Zend\Form\Element\Number',
        'attributes' => array(
                'id' => 'age',
                'min' => 18,
                'max' => 99,
                'step' => 1
        ),
        'options' => array(
                'label' => 'Wieviel Jahre sind sie alt?'
        ),
));
// [..]
```

Listing 16.12

Depending on the browser and its HTML5 support, it becomes immediately obvious that, when using the rules, not only the check on the server functions well, but erroneous entries are queried directly in the client. Indeed, the configuration in the `attributes` array is not only subsequently considered by the appropriate view helper during the generation of the HTML code, but is also used by the validators.

Incidentally, the `Number` element also simultaneously attaches a `Zend\Filter\StringTrim` such that this no longer has to occur in one's own code.

Fieldsets

The more attentive among us have perhaps already noticed that we have not completely recreated the original webform, which we generated manually, with our object version. The fieldset is missing. At this time you should already realise that when I use the term "fieldsets" in the following, I do not explicitly mean the previously mentioned, lacking HTML fieldset, but rather "fieldsets" in the sense of the Zend Framework definition, which initially must not fundamentally have anything to do with HTML fieldsets, but can refer to them, as required. This differentiation is crucially important for your comprehension.

A "fieldset" serves to group individual fields. This is particularly appropriate when a form is composed of elements having different requirements. For example, if we were to expand the form developed above for the address of the user, the data would probably be administered in an entity of its own, stored in a database table of its own, and referenced by the "user" entity or table proper. The technical advantage of fieldsets is the fact that they can be reused in different forms. If we stick to our example, this means that the user would state his or her address in the scope of the newsletter registration (let's just pretend that this would be appropriate), but could also be subsequently adapted to the customer account via the Address-Change Form. In both cases, the fieldset that was defined once would be used. Let's look at a fieldset definition for both fields of the form in an exemplary manner:

```php
<?php
namespace Helloworld\Form;

use Zend\Form\Fieldset;

class UserFieldset extends Fieldset
{
        public function __construct()
        {
                parent::__construct('user');

                $this->add(array(
                        'name' => 'name',
                        'attributes' => array(
                                'type'  => 'text',
                                'id' => 'name'
                        ),
                        'options' => array(
                                'id' => 'name',
                                'label' => 'Ihr Name:',
                        )
                ));

                $this->add(array(
                        'name' => 'email',
                        'attributes' => array(
                                'type'  => 'email',
                        ),
                        'options' => array(
                                'id' => 'email',
                                'label' => 'Ihre E-Mail-Adresse:'
                        ),
                ));
        }
}
```

Listing 16.13[65]

The definition of the fieldset has great similarity with that of the form developed earlier, except for the fact that the fieldset is no longer presented alone as a webform for external purposes, but instead has to be incorporated into a `Zend\Form` in order to be displayed:

[65]https://gist.github.com/3922738

```php
<?php
namespace Helloworld\Form;

use Zend\Form\Form;

class SignUp extends Form
{
        public function __construct()
        {
                parent::__construct('signUp');
                $this->setAttribute('action', '/signup');
                $this->setAttribute('method', 'post');
                $this->setInputFilter(new \Helloworld\Form\SignUpFilter());

                $this->add(new \Helloworld\Form\UserFieldset());

                $this->add(array(
                        'name' => 'submit',
                        'attributes' => array(
                                'type'  => 'submit',
                                'value' => 'Eintragen'
                        ),
                ));
        }
}
```

Listing 16.14[66]

Thus, here is theSignUp Form again, but this time with the UserFieldset, instead of its individual fields, which are now grouped in the UserFieldset. To ensure that the form will still be correctly displayed, we still have to adapt the view code:

```php
<?php
$this->form->prepare();
echo $this->form()->openTag($this->form);
echo $this->formRow($this->form->get('user')->get('name'));
echo $this->formRow($this->form->get('user')->get('email'));
echo $this->formSubmit($this->form->get('submit'));
echo $this->form()->closeTag();
```

Listing 16.15

[66]https://gist.github.com/3922739

We now initially access the fieldset via `get('user')`and from there access the elements of the fieldset. If we do not make this adjustment now, we have made an error. So far so good. But we are still not yet completely finished, because in as much as our form is now no longer correctly validated because the `SignUpFilter` is still assigned to the webform, but the configurations there are no longer appropriate.

```php
<?php
// [..]
$this->setInputFilter(new \Helloworld\Form\SignUpFilter());
// [..]
```

Listing 16.16

Now, we must thus ensure that the `UserFieldset` itself has the required configuration. We do that by providing the `UserFieldset` with the `getInputFilterSpecification()` method and make the required configurations there:

```php
<?php
namespace Helloworld\Form;

use Zend\Form\Fieldset;

class UserFieldset extends Fieldset
{
        public function __construct()
        {
                parent::__construct('user');

                $this->add(array(
                        'name' => 'name',
                        'attributes' => array(
                                'type'  => 'text',
                                'id' => 'name'
                        ),
                        'options' => array(
                                'id' => 'name',
                                'label' => 'Ihr Name:',
                        )
                ));

                $this->add(array(
                        'name' => 'email',
                        'attributes' => array(
                                'type'  => 'email',
                        ),
                        'options' => array(
                                'id' => 'email',
                                'label' => 'Ihre E-Mail-Adresse:'
                        ),
                ));
        }

        public function getInputFilterSpecification()
        {
                return array(
                        'email' => array(
                                'validators' => array(
                                        array(
                                                'name' => 'NotEmpty',
                                                'break_chain_on_failure' => true,
                                                'options' => array(
                                                        'messages' => array(
                                        \Zend\Validator\NotEmpty::IS_EMPTY =>
                                        'Bitte geben Sie etwas ein.'
                                                        )
                                                )
                                        ),
                                        array(
                                                'name' => 'EmailAddress',
                                                'options' => array(
                                                        'messages' => array(
                                        \Zend\Validator\EmailAddress::INVALID_FORMAT
                                        => 'Bitte richtige E-Mail-Adresse eingeben.'
                                                        )
                                                )
                                        ),
                                ),
                        ),
                        'name' => array(
                                'filters' => array(
                                        array(
                                                'name' => 'StringTrim'
                                        )
                                ),
                                'validators' => array(
                                        array(
                                                'name' => 'NotEmpty',
                                                'options' => array(
                                                        'messages' => array(
                                        \Zend\Validator\NotEmpty::IS_EMPTY =>
                                        'Bitte geben Sie etwas ein.'
                                                        )
                                                )
                                        )
                                )
                        )
                );
        }
}
```

Listing 16.17[67]

So, what have we now accomplished here? We took the definitions out of the `SignUpFilter` and transferred them directly into the `getInputFilterSpecification()` method of the fieldset. This is necessary because the fieldset's supraordinate form receives all of its "InputFilter" specifications from the `getInputFilterSpecification()` methods of the referenced fieldsets.

Now, we obtain the following, desired result if the form has been sent with valid inputs.

```
array(2) {
        'submit' => string(9) "Eintragen"
        'user' => array(2) {
                'name' => string(13) "Michael Romer"
                'email' => string(24) "zf2buch@michael-romer.de"
        }
}
```

Linking entities with Forms

As a rule, an application's webforms are related to its entities. A product together with information as to quantity is placed in the shopping basket and thus an order is generated, a user is registered on login, the shipping address is registered in the scope of the checkout, etc. For this reason one is frequently engaged in transferring the validated data from a webform to the appropriate entity, which then, for example, persists in the database. Or, the other way around, data from the database are transferred to an entity in a webform, for example, to make its characteristics editable there.

[67]https://gist.github.com/3922744

To simplify this procedure, so-called “hydrators”, which we have already become acquainted with in the scope of our work with`Zend\Db`, are used. Thus, one could say that we have now come full circle. Initially, we need the appropriate `User` entity for the form:

```php
<?php
namespace Helloworld\Entity;

class User
{
        protected $id;
        protected $email;
        protected $name;

        public function setEmail($email)
        {
                $this->email = $email;
        }

        public function getEmail()
        {
                return $this->email;
        }

        public function setId($id)
        {
                $this->id = $id;
        }

        public function getId()
        {
                return $this->id;
        }

        public function setName($name)
        {
                $this->name = $name;
        }

        public function getName()
        {
                return $this->name;
        }
}
```

Listing 16.18[68]

In order for this entity to be used as a “data container” by the `SignUp` form, one must perform the required configuration for this at the corresponding location. In this example, we now initially again work without fleldsets. Thus, we are dealing with a “normal” form in which the individual elements are actually located.

```php
<?php
namespace Helloworld\Form;

use Zend\Form\Form;

class SignUp extends Form
{
        public function __construct()
        {
                parent::__construct('signUp');
                $this->setAttribute('action', '/signup');
                $this->setAttribute('method', 'post');

                $this->add(array(
                        'name' => 'name',
                        'attributes' => array(
                                'type'  => 'text',
                                'id' => 'name'
                        ),
                        'options' => array(
                                'id' => 'name',
                                'label' => 'Ihr Name:',
                        )
                ));

                $this->add(array(
                        'name' => 'email',
                        'attributes' => array(
                                'type'  => 'email',
                        ),
                        'options' => array(
                                'id' => 'email',
                                'label' => 'Ihre E-Mail-Adresse:'
                        ),
                ));
```

[68]https://gist.github.com/3923101

```
        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type'  => 'submit',
                'value' => 'Eintragen'
            ),
        ));
    }
}
```

Listing 16.19[69]

For simplicity's sake, we will neglect the definitions of validators and filters at this time. We now link an entity in the controller and configure it to the employed hydrator:

```php
<?php

namespace Helloworld\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class IndexController extends AbstractActionController
{
    public function indexAction()
    {
        $form = new \Helloworld\Form\SignUp();
        $form->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
        $form->bind(new \Helloworld\Entity\User());

        if ($this->getRequest()->isPost()) {
            $form->setData($this->getRequest()->getPost());

            if ($form->isValid()) {
                var_dump($form->getData());
            } else {
                return new ViewModel(
                    array(
                        'form' => $form
                    )
```

[69] https://gist.github.com/3923105

```
                                );
                        }
                } else {
                        return new ViewModel(
                                array(
                                        'form' => $form
                                )
                        );
                }
        }
}
```

Listing 16.20[70]

If we now send the webform with validated data, a completely filled entity is returned via `$form->getData()` (instead of an array as was previously the case):

```
class Helloworld\Entity\User#186 (3) {
        protected $id => NULL
        protected $email => string(24) "zf2buch@michael-romer.de"
        protected $name => string(13) "Michael Romer"
}
```

The essential configuration is the invocation of `bind()`, in which we consign the object (or its class, respectively) to the form, into which the data is to be transferred with the aid of the previously defined `Reflection` hydrator. The following "hydrators" are included in Framework as standard.

- `ArraySerializable`: This is the standard hydrator, which `Zend\Form` uses if nothing else has been defined. It expects that the respective object implements the `getArrayCopy()` and `exchangeArray()` or `populate()`, respectively, and makes the required information available in this manner.
- `ClassMethods`: uses the object's Getter/Setter methods (or the class, respectively) to insert (`hydrate()`) or read out (`extract()`), respectively, the appropriate data..
- `ObjectProperty`: uses the object's public properties.
- `Reflection`: uses PHP's `ReflectionClass` to determine the object's properties and to set or read out the appropriate values. Since this hydrator makes use of `$property->setAccessible(true)`, `private` properties can also be managed in this manner.

And now let's look at everything again using fleldsets because there is also a useful feature in this context when one works with several objects which reference one another. To begin with, we create

[70] https://gist.github.com/3923106

another entity, which we call “UserAddress” and in which we map the user’s address, which we would like to query directly in the scope of the login. The data should be managed in the application, but also be independently treated as an entity and also subsequently be stored in the database in a table of their own.

```php
<?php
namespace Helloworld\Entity;

class UserAddress
{
        private $street;
        private $streetNumber;
        private $zipcode;
        private $city;

        public function setStreet($street)
        {
                $this->street = $street;
        }

        public function getStreet()
        {
                return $this->street;
        }

        public function setCity($city)
        {
                $this->city = $city;
        }

        public function getCity()
        {
                return $this->city;
        }

        public function setStreetNumber($streetNumber)
        {
                $this->streetNumber = $streetNumber;
        }

        public function getStreetNumber()
        {
                return $this->streetNumber;
```

```php
	}

	public function setZipcode($zipcode)
	{
		$this->zipcode = $zipcode;
	}

	public function getZipcode()
	{
		return $this->zipcode;
	}
}
```

Listing 16.21[71]

We extend the `User` entity by the `$userAddress` property, which symbolises the reference to the appropriate `UserAddress` entity.

```php
<?php
namespace Helloworld\Entity;

class User
{
	protected $id;
	protected $email;
	protected $name;
	protected $userAddress;

	public function setEmail($email)
	{
		$this->email = $email;
	}

	public function getEmail()
	{
		return $this->email;
	}

	public function setId($id)
	{
		$this->id = $id;
```

[71] https://gist.github.com/3924042

```
        }

        public function getId()
        {
                return $this->id;
        }

        public function setName($name)
        {
                $this->name = $name;
        }

        public function getName()
        {
                return $this->name;
        }

        public function setUserAddress($userAddress)
        {
                $this->userAddress = $userAddress;
        }

        public function getUserAddress()
        {
                return $this->userAddress;
        }
}
```

Listing 16.22[72]

Additionally, we generate the new `UserAddressFieldset` together with the required filter specification, which we are already familiar with, as well as the reference to the hydrator that is to be used and the appropriate entity, into which this fieldset's data is to be transferred.

[72]https://gist.github.com/3924043

```
<?php
namespace Helloworld\Form;

use Zend\Form\Fieldset;

class UserAddressFieldset extends Fieldset
{
        public function __construct()
        {
                parent::__construct('userAddress');
                $this->setHydrator(new \Zend\Stdlib\Hydrator\Reflection());
                $this->setObject(new \Helloworld\Entity\UserAddress());

                $this->add(array(
                        'name' => 'street',
                        'attributes' => array(
                                'type'  => 'text',
                        ),
                        'options' => array(
                                'label' => 'Ihre Strasse:',
                        )
                ));

                $this->add(array(
                        'name' => 'streetNumber',
                        'attributes' => array(
                                'type'  => 'text',
                        ),
                        'options' => array(
                                'label' => 'Ihre Hausnummer:',
                        )
                ));

                $this->add(array(
                        'name' => 'zipcode',
                        'attributes' => array(
                                'type'  => 'text',
                        ),
                        'options' => array(
                                'label' => 'Ihre Postleitzahl:',
                        )
                ));

                $this->add(array(
                        'name' => 'city',
                        'attributes' => array(
                                'type'  => 'text',
                        ),
                        'options' => array(
                                'label' => 'Ihre Stadt:',
                        )
                ));
        }
}
```

Listing 16.23[73]

We incorporate the `UserAddressFieldset` appropriately, but not directly in the `SignUp` form, but “nested” in the `UserFieldSet` instead:

```
<?php
namespace Helloworld\Form;

use Zend\Form\Fieldset;

class UserFieldset extends Fieldset
{
        public function __construct()
        {
                parent::__construct('user');

                $this->add(array(
                        'name' => 'name',
                        'attributes' => array(
                                'type'  => 'text',
                                'id' => 'name'
                        ),
                        'options' => array(
                                'id' => 'name',
                                'label' => 'Ihr Name:',
                        )
                ));

```

[73]https://gist.github.com/3924047

```
                $this->add(array(
                        'name' => 'email',
                        'attributes' => array(
                                'type'  => 'email',
                        ),
                        'options' => array(
                                'id' => 'email',
                                'label' => 'Ihre E-Mail-Adresse:'
                        ),
                ));

                $this->add(array(
                        'type' => 'Helloworld\Form\UserAddressFieldset',
                        )
                );
        }
}
```

Listing 16.24[74]

The actual form now looks like this:

```
<?php
namespace Helloworld\Form;

use Zend\Form\Form;

class SignUp extends Form
{
        public function __construct()
        {
                parent::__construct('signUp');
                $this->setAttribute('action', '/signup');
                $this->setAttribute('method', 'post');

                $this->add(array(
                        'type' => 'Helloworld\Form\UserFieldset',
                        'options' => array(
                                'use_as_base_fieldset' => true
                        )
                ));
```

[74]https://gist.github.com/3924051

```

                $this->add(array(
                        'name' => 'submit',
                        'attributes' => array(
                                'type'  => 'submit',
                                'value' => 'Eintragen'
                        ),
                ));
        }
}
```

Listing 16.25[75]

In this case, the important thing is the `'use_as_base_fieldset' => true`, so that the assignment of values and object properties functions correctly, and proceeding from the `UserFieldset` that the data are distributed across the appropriate entities.

And finally, don't forget to adjust the fleldset's view file appropriately such that the fields are all correctly displayed and that no errors occur:

```php
<?php
$this->form->prepare();
echo $this->form()->openTag($this->form);

echo $this->formRow($this->form->get('user')
        ->get('name'));

echo $this->formRow($this->form->get('user')
        ->get('email'));

echo $this->formRow($this->form->get('user')
        ->get('userAddress')->get('street'));

echo $this->formRow($this->form->get('user')
        ->get('userAddress')->get('streetNumber'));

echo $this->formRow($this->form->get('user')
        ->get('userAddress')->get('zipcode'));

echo $this->formRow($this->form->get('user')
        ->get('userAddress')->get('city'));

```

[75]https://gist.github.com/3924059

```
echo $this->formSubmit($this->form->get('submit'));
echo $this->form()->closeTag();
```

Listing 16.26[76]

If we now send the form with valid data, we no longer receive only the `User` object, but also the filled-out, referenced `UserAddress` object:

```
class Helloworld\Entity\User#201 (4) {
        protected $id => NULL
        protected $email => string(24) "zf2buch@michael-romer.de"
        protected $name => string(13) "Michael Romer"
        protected $userAddress =>
                class Helloworld\Entity\UserAddress#190 (4) {
                        private $street => string(14) "Grevingstrasse"
                        private $streetNumber => string(2) "35"
                        private $zipcode => string(5) "48151"
                        private $city => string(8) "Münster"
                }
}
```

Further simplify processing by means of annotations

The sketched-out path for the processing of forms has already made the application developer's life distinctly easier. Nevertheless, a good deal of code is still required to perform the configurations. An alternative way is the use of annotations in the entity classes, from which a large part of the otherwise manually generated configuration can be dynamically generated by Framework. Further information on this topic can be found, as needed, in the official Framework documentation.

[76]https://gist.github.com/3924070

Developers’ Dairy

Introduction

Up to this point, we mainly talked about the framework’s core concepts and its individual components. Now, we will jump straight into an example application and tackle the challenges of the day-to-day business with Zend Framework 2.

For development of the example application we will use the “Scrum” method, whenever possible. Scrum is an iterative and incremental agile software development framework for managing software projects and product or application development. I adopted agile some years ago and Scrum specifically around 2008. I think Scrum really is helping mastering the art of professional software development and it helped me a lot in my professional projects. This is why I highly recommend considering Scrum whenever possible. It’s simply not enough to only write good code e.g. by using Zend Framework 2 to succeed in a complex software project. You will also need to organize yourself.

We start with the so called “Envisioning” which helps shaping the idea of the final product we’ll develop then with each so called “Sprints”; fixed windows of development time.

Envisioning

In the following, we want to implement the core functionality of “ZfDeals”. ZfDeals is an application for selling products online to special reduced prices. The idea is to not develop a stand-alone application, but a ZF2 module that may be used by others in their applications. For demonstration and development purposes, we will come up with a sample “host application” for the module as well.

Sprint 1 - Code Repository, Development Environment and the initial Codebase

The first Sprint is meant to get myself ready for development.

Set up git repository and first commit

First, I need a code repository to maintain my code. I will want to switch between versions, branch and merge and simply know my code managed well. I choose git locally as well as GitHub as my external repository I can push code to regularly. I already have a GitHub account, so I don’t need to register again. However, it only takes minutes to sign up and by the way it’s for free if one uses it for open source projects. Nice!

Git is already installed on my local box, so I can do

```
$ git clone https://github.com/zendframework/
        ZendSkeletonApplication.git ZfDealsApp
```

in a directory of my choice to clone the "ZendSkeletonApplication" that will act as the starting point for my individual coding.

Composer downloads Zend Framework 2 into the `vendor` directory simply by executing:

```
$ cd ZfDealsApp
$ php composer.phar install
```

Now, I set up a new git repository on GitHub. It acts as a code backup to my local respository as well it makes sharing the code with others easy. First, I remove the current origin reference from my local repository

```
$ git remote rm origin
```

and I replace it with the new GitHub repository:

```
$ git remote add origin https://github.com/michael-romer/ZfDealsApp.git
```

A simple

```
$ git push -u origin master
```

pushes the code to the repository on GitHub[77].

Local development environment

Instead of setting up Apache, PHP and all other software on my local box, I create a virtual machine acting as my local runtime environment. I use a code library available in my GitHub-Account[78]. Again Composer helps to download and install it by adding it as a dependency to my `composer.json`:

[77]https://github.com/michael-romer/ZfDealsApp
[78]https://github.com/michael-romer/zfb2-vm

```
{
        "name": "zendframework/skeleton-application",
        "description": "Skeleton Application for ZF2",
        "license": "BSD-3-Clause",
        "keywords": [
            "framework",
            "zf2"
        ],
        "homepage": "http://framework.zend.com/",
        "require": {
            "php": ">=5.3.3",
            "zendframework/zendframework": "dev-master",
            "zfb/zfb-vm": "dev-master"
        }
}
```

Executing

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

on the shell then downloads the library. Last but not least I need to copy and rename `/vendor/zfb/zfb-vm/Vagrantf` to `/Vagrantfile` and install some tools on my local box:

- Virtual Box[79]
- Ruby[80]
- Vagrant[81]

Yes, it's a bit of work, however, it's a one-time-effort and once done, one can easily spin up new virtual machines capable of running Zend Framework 2 applications in a matter of minutes. This is helpful e.g. if you work on different projects at the same time, especially, if a slightly different runtime configuration is needed, such as a different PHP version. Furthermore, setting up the runtime environment on my local box would not have been faster anyway.

Now, two more shell commands before going to an extended lunch:

```
$ vagrant box add precise64 http://files.vagrantup.com/precise64.box
$ vagrant up
```

Once the virtual machine has been created, one may open `localhost:8080`.

Via

[79]https://www.virtualbox.org/wiki/Downloads
[80]http://www.ruby-lang.org/de/
[81]http://vagrantup.com/

```
$ vagrant ssh
```

or by using Putty if you have a windows box[82], one may connect to the virtual machine's shell (use `exit` to get out again). On the virtual machine, the `/vagrant`-folder is shared between the host system and the virtual machine, allowing you to write code within the IDE of your choice on your local box and at the same time put the code into execution on the virtual machine's LAMP stack. As you my have noticed, port 8080 of your local box has been configured to forward to port 80 of the virtual machine.

If you stop developing for a while, you may run

```
$ vagrant suspend
```

to put the virtual machine into standby. You can wake it up again running

```
$ vagrant resume
```

If you don't need the VM for longer, you may halt the system via

```
$ vagrant halt
```

and restart it via

```
$ vagrant up
```

Restarting a stopped VM takes longer than resuming a suspended one.

For sure, a virtual machine is not needed to work with Zend Framework 2. It absolutely fine to use XAMPP[83] or set up your own LAMP-like stack by hand directly on your box. Just make sure at some point you have a proper configured PHP 5.3.3 web-environment available to run your code in. Make sure also to configure you webserver to treat the application's `public` directory as its "Document Root". This is where the `index.php` file is stored which serves as the "main entrance" for all functions of a ZF2 application.

At some point you hopefully will then see

```
"Welcome to Zend Framework 2"
```

on your screen. It means you successfully finished Sprint 1!

[82]http://vagrantup.com/v1/docs/getting-started/ssh.html
[83]http://www.apachefriends.org/xampp.html

Sprint 2 - A custom module with add product functionality

Source Code download

The source code of Sprint 2 can be downloaded using Tag "Sprint2" on GitHub[a].

[a] https://github.com/michael-romer/ZfDealsApp/tree/Sprint2

User Stories

While Sprint 1 was focused on setting up the initial codebase and the development environment, Sprint 2 brings the first functional requirement onto the table: Adding new products to the system that may be sold later at a special discount price. In Scrum, requirements are usually given by using a technique called "user stories":

> "In software development and product management, a user story is one or more sentences in the everyday or business language of the end user or user of a system that captures what a user does or needs to do as part of his or her job function." (Wikipedia)

Usually, the sentences follow the pattern:

> "In order to [receive benefit] as a [role], I want [goal/desire]"

Therefore, the first requirement reads as follows:

> "In order to offer a product with a special discount price as a merchant, I want to add product details to the system."

This sounds reasonable. Usually, in addition the "user story sentence" one will want to state so called acceptance criteria that go into more detail on the requirements:

- One may add a unique ID, description and stock information per product.
- One may add all product data using a web form.

Create a custom module

So, let's go! First, I add a new custom ZF2 module that holds all functionality of ZfDeals. I set up the following directory and file structure in `module`:

```
ZfDeals/
        Module.php
        config/
                module.config.php
        src/
                ZfDeals/
                        Controller/
                                AdminController.php
        view/
                zf-deals/
                        admin/
                                index.phtml
                layout/
                        admin.phtml
```

ZFTool

Instead of setting up the directory and file structure all by hand, you could use ZFTool[a] and let it create most of the directories and files automatically. However, ZFTool is not bundled with the ZF2 library and must be downloaded separately.

[a] https://github.com/zendframework/ZFTool

In `Module.php` I only add the most basic code for autoloading the module's classes and pointing the framework's `ModuleManager` to the module's config file:

```php
<?php
namespace ZfDeals;

class Module
{
        public function getConfig()
        {
                return include __DIR__ . '/config/module.config.php';
        }

        public function getAutoloaderConfig()
        {
                return array(
                        'Zend\Loader\StandardAutoloader' => array(
                                'namespaces' => array(
                                        __NAMESPACE__
                                                => __DIR__ . '/src/' . __NAMESPACE__,
                                ),
                        ),
                );
        }
}
```

Listing 26.1

My bare new `AdminController` is based on `AbstractActionController`, allowing to work with custom "actions":

```php
<?php
namespace ZfDeals\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class AdminController extends AbstractActionController
{
        public function indexAction()
        {
                return new ViewModel();
        }
}
```

Listing 26.2

The file `module.config.php` is straightforward. It holds a literal route to the ZfDeals' admin section homepage

```php
<?php
return array(
    'router' => array(
                'routes' => array(
                        'zf-deals\admin\home' => array(
                                'type' => 'Zend\Mvc\Router\Http\Literal',
                                'options' => array(
                                        'route'    => '/deals/admin',
                                        'defaults' => array(
                                                'controller'
```

```
                                                         => 'ZfDeals\Controller\Admin',
                                               'action'
                                                         => 'index',
                                          ),
                                   ),
                            ),
                     ),
              ),
              // [..]
)
```

Listing 26.3

as well as the `AdminController` declaration

```
<?php
// [..]
'controllers' => array(
    'invokables' => array(
        'ZfDeals\Controller\Admin'
                => 'ZfDeals\Controller\AdminController'
    ),
),
// [..]
```

Listing 26.4

and the `admin` layout declaration. In class `Module` I make sure that this layout is used whenever the `AdminController` is dispatched:

```
<?php
// [..]
public function init(\Zend\ModuleManager\ModuleManager $moduleManager)
{
    $sharedEvents = $moduleManager
            ->getEventManager()->getSharedManager();
    $sharedEvents->attach(
            'ZfDeals\Controller\AdminController',
            'dispatch',
                function($e) {
                        $controller = $e->getTarget();
                        $controller->layout('zf-deals/layout/admin');
                },
                100
        );
}
// [..]
```

Listing 26.5

I register a callback function for the `dispatch` event in method `init`. The `dispatch` event is emitted by the `ZfDeals\Controller\AdminController` when dispatched so the callback is executed. The reason the `SharedEventManager` is used here is simple: At the point time, when I attach my event handler, the `AdminController` itself with its composed own `EventManager` has not been instantiated yet. So I will need to to go through the `SharedEventManager` here to attach my handler.

The callback function itself uses the `layout` controller plugin to set the layout to the one defined in `module.config.php`:

```
<?php
// [..]
'view_manager' => array(
    'template_map' => array(
        'zf-deals/layout/admin' => __DIR__ . '/../view/layout/admin.phtml',
    ),
    'template_path_stack' => array(
        __DIR__ . '/../view',
    ),
),
// [..]
```

Listing 26.6

Don't forget to activate the new module by adding it to the list of modules in `application.config.php`:

```
<?php
return array(
        'modules' => array(
                'Application',
                'ZfDeals'
        ),
        // [..]
);
```

Listing 26.7

After adding an empty action method called `index` to the new controller as well as creating a view file, I can open `/deals/admin` in a browser and see the admin layout design on the screen.

Dealing with static assets

At this stage, I'm asking myself how to deal with static assets my module will contain, like images such as the ZfDeals-logo, css and js files. If one wants to serve assets to a client, in general, they need to be publicly available, e.g. by putting them somewhere in the `public`-folder of the host application. This means, if I ship my assets bundled with the ZfDeals module, they won't be available by default. So, what to do? Unfortunately, ZF2 does not bring any support "out-of-the-box" and one needs to be creative to solve this issue. Here are the options:

- I develop a controller within my module, that serves static assets via PHP by using `file_-get_contents()` or similar. This probably will work fine, however, it isn't very smart to add such "infrastructure" code to ZfDeals and if I don't implement a mechanism of caching, it will surely become a performance bottleneck at some stage.
- I use an asset manager library for ZF2 such as AssetManager[84] and add this module to the list of dependencies of ZfDeals. However, this would require me to install another module besides ZfDeals itself to make things work properly.
- I simply copy all assets over to the `public` directory of the host application. However, this means, when distributing the module, I will need to instruct the developer on how to integrate ZfDeal in his own application by copying over files manually.
- I simply copy the assets over to the `public` directory of the host application. But instead of putting them directly into the `public` directory, I add another directory named after the module first. However, this means, when distributing the module, I will need to instruct the developer on how to integrate ZfDeal in his own application by copying over files manually.
- I keep the assets within the module, but add a symlink to the `public` directory. Again, this means, when distributing the module, I will need to instruct the developer on how to integrate ZfDeal in his own application by setting up a symlink.

I guess utilizing a proper asset manager will be my first choice in the long run, however, for now, I will live with copying over files to the host application's public directory.

Web form for adding a product

Now I add a first web form to ZfDeals. The form is used to add new products to the system. Instead of putting product related fields directly into the form, I encapsulate them in a fieldset. The fieldset is added to the form then:

[84]https://github.com/RWOverdijk/AssetManager

```php
<?php
namespace ZfDeals\Form;

use Zend\Form\Form;

class ProductAdd extends Form
{
        public function __construct()
        {
                parent::__construct('login');
                $this->setAttribute('action', '/deals/admin/product/add');
                $this->setAttribute('method', 'post');

                $this->add(array(
                        'type' => 'ZfDeals\Form\ProductFieldset',
                        'options' => array(
                                'use_as_base_fieldset' => true
                        )
                ));

                $this->add(array(
                        'name' => 'submit',
                        'attributes' => array(
                                'type'  => 'submit',
                                'value' => 'Hinzufügen'
                        ),
                ));
        }
}
```

Listing 26.8

The form `ProductAdd` is composed of an "add" button as well as the `ProductFieldset`, which also holds the filter and validator definitions:

```php
<?php
namespace ZfDeals\Form;

use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterInterface;
use Zend\InputFilter\InputFilterProviderInterface;

class ProductFieldset extends Fieldset
        implements InputFilterProviderInterface
{
        public function __construct()
        {
                parent::__construct('product');

                $this->add(array(
                        'name' => 'id',
                        'attributes' => array(
                                'type'  => 'text',
                ),
                        'options' => array(
                                'label' => 'Produkt-ID:',
                        )
                ));


                $this->add(array(
                        'name' => 'name',
                        'attributes' => array(
                                'type'  => 'text',
                        ),
                        'options' => array(
                                'label' => 'Produktbezeichnung:',
                        )
                ));

                $this->add(array(
                        'name' => 'stock',
                        'attributes' => array(
                                'type'  => 'number',
                        ),
                        'options' => array(
                                'label' => '# Bestand:'
                        ),
                ));
        }

        public function getInputFilterSpecification()
        {
                return array(
                        'id' => array (
                                'required'   => true,
                                'filters' => array(
                                        array(
                                                'name' => 'StringTrim'
                                        )
                                ),
                                'validators' => array(
                                        array(
                                                'name' => 'NotEmpty',
                                                'options' => array(
                                                        'message'  =>
                                        "Bitte geben Sie die Produkt-ID an."
                                                )
                                        )
                                )
                        ),
                        'name' => array (
                                'required'   => true,
                                'filters' => array(
                                        array(
                                                'name' => 'StringTrim'
                                        )
                                ),
                                'validators' => array(
                                        array(
                                                'name' => 'NotEmpty',
                                                'options' => array(
                                                        'message'  =>
                                        "Bitte geben Sie eine Produktbezeichnung an."
                                                ),
                                        )
                                )
                        ),
                        'stock' => array (
                                'required'   => true,
                                'filters' => array(
                                        array(
                                                'name' => 'StringTrim'
                                        )
                                ),
                                'validators' => array(
                                        array(
                                                'name' => 'NotEmpty',
                                                'options' => array(
                                                        'message'  =>
                                        "Bitte geben Sie die Lagerbestand an."
                                                )
                                        ),
                                        array(
                                                'name' => 'Digits',
                                                'options' => array(
                                                        'message'  =>
                                        "Bitte geben Sie einen ganzzahligen Wert an."
                                                )
                                        ),
                                        array(
                                                'name' => 'GreaterThan',
                                                'options' => array(
                                                        'min' => 0,
                                                        'message'  =>
                                        "Bitte geben Sie Wert >= 0 an."
                                                )
                                        )
                                )
                        )
                );
        }
}
```

Listing 26.9

This approach allows to re-use the product field definitions in other forms, e.g. a form that is dedicated to editing an existing product.

Display the form

An additional route and action take care of displaying and processing the form:

```php
<?php
return array(
        'router' => array(
                'routes' => array(
                        'zf-deals\admin\home' => array(
                                'type' => 'Zend\Mvc\Router\Http\Literal',
                                'options' => array(
                                        'route'    => '/deals/admin',
                                        'defaults' => array(
                                                'controller'
                                                        => 'ZfDeals\Controller\Admin',
                                                'action'
                                                        => 'index',
                                        ),
                                ),
                        ),
                        'zf-deals\admin\product\add' => array(
                                'type' => 'Zend\Mvc\Router\Http\Literal',
                                'options' => array(
                                        'route'    => '/deals/admin/product/add',
                                        'defaults' => array(
                                                'controller'
                                                        => 'ZfDeals\Controller\Admin',
                                                'action'
                                                        => 'add-product',
                                        )
                                )
                        )
                )
        )
        // [..]
)
```

Listing 26.10

The action itself reads as follows:

```php
<?php
// [..]
public function addProductAction()
{
        $form = new \ZfDeals\Form\ProductAdd();

        if ($this->getRequest()->isPost()) {
                $form->setData($this->getRequest()->getPost());

                if ($form->isValid()) {
                        // todo
                } else {
                        return new ViewModel(
                                array(
                                        'form' => $form
                                )
                        );
                }
        } else {
                return new ViewModel(
                        array(
                                'form' => $form
                        )
                );
        }
}
// [..]
```

Listing 26.11

In the corresponding view I display the form:

```php
<?php
$this->form->prepare();
echo $this->form()->openTag($this->form);
echo $this->formRow($this->form->get('product')->get('id'));
echo $this->formRow($this->form->get('product')->get('name'));
echo $this->formRow($this->form->get('product')->get('stock'));
echo $this->formSubmit($this->form->get('submit'));
echo $this->form()->closeTag();
```

Listing 26.12

If I now open `/deals/admin/product/add` I can already see the form being rendered. However, it doesn't look that pretty yet. I can make it look more beautiful by adding "Twitter Bootstrap[85]" and instead of adding all required markup by hand, I opt for adding another ZF2 module: "DluTwBootstrap[86]". As always, I utilize Composer to install and configure the module by adding a dependency:

```
"dlu/dlutwbootstrap": "dev-master"
```

Then I run

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

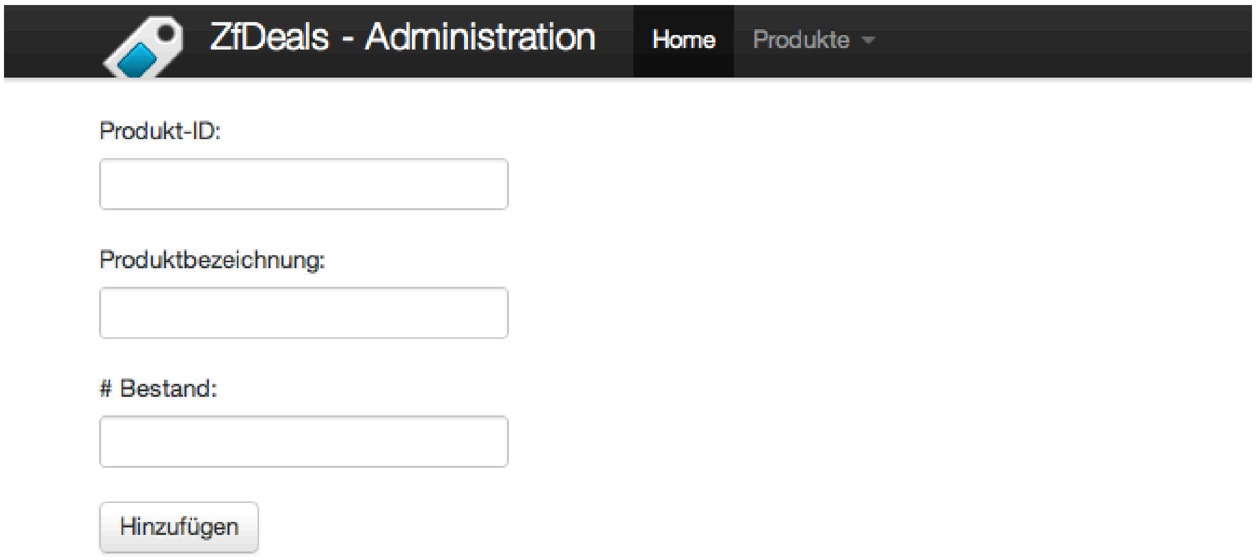
and the module is being installed (do not forget to activate it in `application.config.php`!). The module mainly registers a bunch of additional "View Helper" for displaying the form using Twitter Bootstrap:

```php
<?php
$this->form->prepare();
echo $this->form()->openTag($this->form);
echo $this->formRowTwb($this->form->get('product')->get('id'));
echo $this->formRowTwb($this->form->get('product')->get('name'));
echo $this->formRowTwb($this->form->get('product')->get('stock'));
echo $this->formSubmitTwb($this->form->get('submit'));
echo $this->form()->closeTag();
```

Listing 26.13

And this how it looks right now:

[85]http://twitter.github.com/bootstrap/
[86]https://bitbucket.org/dlu/dlutwbootstrap/overview



ZfDeals - Add product form

Unit-Tests for the web form

Before I start working on getting data out of the form and into the database, I will first add some unit tests for form `ProductAdd`, just to be sure, it's configured correctly. I create a new directory `tests` in the application root and add the file `phpunit.xml` to configure the directories holding test cases:

```
<phpunit bootstrap="./bootstrap.php">
        <testsuites>
                <testsuite name="AllTests">
                        <directory>./ZfDealsTest/FormTest</directory>
                </testsuite>
        </testsuites>
</phpunit>
```

As I need to bootstrap the application with all of its services to actually run tests, I add a proper `bootstrap.php` file to the `tests` directory as well:

```php
<?php
use Zend\Loader\StandardAutoloader;

chdir(dirname(__DIR__));

include 'init_autoloader.php';

$loader = new StandardAutoloader();
$loader->registerNamespace('ZfDealsTest', __DIR__ . '/ZfDealsTest');
$loader->register();

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

Listing 26.14

Its executed automatically by PHPUnit, when running tests. In `bootstrap.php` I configure autoloading of test classes stored in directory `ZfDealsTest`. The test file `ProductAddTest` goes into its subdirectory `FormTest`:

```php
<?php
namespace ZfDealsTest\FormTest;

use ZfDeals\Form\ProductAdd;

class ProductAddTest extends \PHPUnit_Framework_TestCase
{
        private $form;
        private $data;

        public function setUp()
        {
                $this->form = new ProductAdd();
                $this->data = array(
                        'product' => array(
                                'id' => '',
                                'name' => '',
                                'stock' => ''
                        )
                );
        }

        public function testEmptyValues()
        {
                $form = $this->form;
                $data = $this->data;

                $this->assertFalse($form->setData($data)->isValid());

                $data['product']['id'] = 1;
                $this->assertFalse($form->setData($data)->isValid());

                $data['product']['name'] = 1;
                $this->assertFalse($form->setData($data)->isValid());

                $data['product']['stock'] = 1;
                $this->assertTrue($form->setData($data)->isValid());
        }

        public function testStockElement()
        {
                $form = $this->form;
                $data = $this->data;
                $data['product']['id'] = 1;
                $data['product']['name'] = 1;

                $data['product']['stock'] = -1;
                $this->assertFalse($form->setData($data)->isValid());

                $data['product']['stock'] = "test";
                $this->assertFalse($form->setData($data)->isValid());

                $data['product']['stock'] = 12.3;
                $this->assertFalse($form->setData($data)->isValid());

                $data['product']['stock'] = 12;
                $this->assertTrue($form->setData($data)->isValid());
        }
}
```

Listing 26.15

To test the form, I hand in different combinations of test data and validate the form's behavior.

Set up the product entity

I now start modeling the so called "Business Domain" by adding the `Product` entity class representing a product in the system:

```php
<?php
namespace ZfDeals\Entity;

class Product
{
        protected $id;
        protected $name;
        protected $stock;

        public function setName($name)
        {
                $this->name = $name;
        }

        public function getName()
        {
                return $this->name;
        }

        public function setId($id)
        {
                $this->id = $id;
        }

        public function getId()
        {
                return $this->id;
        }

        public function setStock($stock)
        {
                $this->stock = $stock;
        }

        public function getStock()
        {
                return $this->stock;
        }
}
```

Listing 26.16

A product entity holds its ID, a description and stock information. The file is added to a directory called `Entity` in `scr/ZfDeals` within my module.

Product entity persistence

The database mapper mapping the entity fields to columns in the database table reads as follows:

```
<?php

namespace ZfDeals\Mapper;

use ZfDeals\Entity\Product as ProductEntity;
use Zend\Stdlib\Hydrator\HydratorInterface;
use Zend\Db\TableGateway\TableGateway;
use Zend\Db\TableGateway\Feature\RowGatewayFeature;
use Zend\Db\Sql\Sql;
use Zend\Db\Sql\Insert;

class Product extends TableGateway
{
        protected $tableName  = 'product';
        protected $idCol = 'id';
        protected $entityPrototype = null;
        protected $hydrator = null;

        public function __construct($adapter)
        {
                parent::__construct($this->tableName,
                        $adapter,
                        new RowGatewayFeature($this->idCol)
                );

                $this->entityPrototype = new ProductEntity();
                $this->hydrator = new \Zend\Stdlib\Hydrator\Reflection;
        }

        public function insert($entity)
        {
                return parent::insert($this->hydrator->extract($entity));
        }
}
```

Listing 26.17

"Convention over configuration" makes the mapper straightforward, if the entity fields match the column names in the database table. Thanks to `\Zend\Stdlib\Hydrator\Reflection` all "mapping magic" mainly happens automatically when calling the `insert` method.

Now let's add the missing database adapter to `module.config.php`. It's needed to actually connect to the database:

```
<?php
// [..]
'service_manager' => array(
        'factories' => array(
                'Zend\Db\Adapter\Adapter' => function ($sm) {
                        $config = $sm->get('Config');
                        $dbParams = $config['dbParams'];

                        return new Zend\Db\Adapter\Adapter(array(
                                'driver' => 'pdo',
                                'dsn' =>
                                        'mysql:dbname='.$dbParams['database']
                                        .';host='.$dbParams['hostname'],
                                'database' => $dbParams['database'],
                                'username' => $dbParams['username'],
                                'password' => $dbParams['password'],
                                'hostname' => $dbParams['hostname'],
                        ));
                }
        )
)
// [..]
```

Listing 26.18

I put the database connection credentials and details in `db.local.php` in directory `/config/autoload`. I do not add this file to the code repository as it contains sensitive data. However, I add another file called `db.local.php.dist` which I commit instead. It acts as a template for the config file that needs to be present on a box running ZfDeals:

```php
<?php
return array(
        'dbParams' => array(
                'database'  => '',
                'username'  => '',
                'password'  => '',
                'hostname'  => '',
        )
);
```

Listing 26.19

This way I make sure that the structure of the database config file is understood by developers integrating ZfDeals into their own applications.

In `ServiceManager` I make `ZfDeals\Mapper\Product` available and inject its dependency to `Zend\Db\Adapter\Adapt`

```php
<?php
// [..]
'service_manager' => array(
        'factories' => array(
                'ZfDeals\Mapper\Product' => function ($sm) {
                        return new \ZfDeals\Mapper\Product(
                                $sm->get('Zend\Db\Adapter\Adapter')
                        );
                },
        )
)
// [..]
```

Listing 26.20

Form processing

I can now add persistence code to `addProductAction()`. I read data from the form and by binding a new product entity object first, calling `getData()` gives back the object automatically populated with the data given. `ZfDeals\Mapper\Product` is then used to save the entity to the database:

```php
<?php
// [..]
public function addProductAction()
{
    $form = new \ZfDeals\Form\ProductAdd();

    if ($this->getRequest()->isPost()) {
        $form->setHydrator(new\Zend\Stdlib\Hydrator\Reflection());
        $form->bind(new \ZfDeals\Entity\Product());
        $form->setData($this->getRequest()->getPost());

        if ($form->isValid()) {
            $newEntity = $form->getData();

            $mapper = $this->getServiceLocator()
                    ->get('ZfDeals\Mapper\Product');

            $mapper->insert($newEntity);
            $form = new \ZfDeals\Form\ProductAdd();

            return new ViewModel(
                array(
                    'form' => $form,
                    'success' => true
                )
            );
        } else {
            return new ViewModel(
                array(
                    'form' => $form
                )
            );
        }
    } else {
        return new ViewModel(
            array(
                'form' => $form
            )
        );
    }
}
// [..]
```

Listing 26.21

But all of this only works after the database table is set up:

```
CREATE TABLE product(
        id varchar(255) NOT NULL,
        name varchar(255) NOT NULL,
        stock int(10) NOT NULL, PRIMARY KEY (id)
);
```

One can now submit the form and a new database entry is added. If it all worked out, a success message is displayed:

```
<?php if ($this->success) { ?>
<div class="alert alert-success">Produkt hinzugefügt!</div>
<?php } ?>

<?php
$this->form->prepare();
echo $this->form()->openTag($this->form);
echo $this->formRowTwb($this->form->get('product')->get('id'));
echo $this->formRowTwb($this->form->get('product')->get('name'));
echo $this->formRowTwb($this->form->get('product')->get('stock'));
echo $this->formSubmitTwb($this->form->get('submit'));
echo $this->form()->closeTag();
```

Listing 26.22

Dependency Injection

So far, so good. However, there is a lot of code that can be improved. Too often, I use the `new` statement in my code making it directly dependent on other classes. This bad practice makes my code hard to test and I want to avoid it whenever possible by applying "dependency injection". This will make testing easier as well as make my code more versatile. Let's do some refactoring right away.

The new `AdminControllerFactory` is added to create the `AdminController`. The factory takes care of injecting the `ZfDeals\Mapper\Product` dependency:

```php
<?php
namespace ZfDeals\Controller;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class AdminControllerFactory implements FactoryInterface
{
        public function createService(ServiceLocatorInterface $serviceLocator)
        {
                $ctr = new AdminController();
                $form = new \ZfDeals\Form\ProductAdd();
                $form->setHydrator(new\Zend\Stdlib\Hydrator\Reflection());
                $form->bind(new \ZfDeals\Entity\Product());
                $ctr->setProductAddForm($form);

                $mapper = $serviceLocator->getServiceLocator()
                        ->get('ZfDeals\Mapper\Product');

                $ctr->setProductMapper($mapper);
                return $ctr;
        }
}
```

Listing 26.23

In `module.config.php` I now point to the factory:

```php
<?php
// [..]
'controllers' => array(
    'factories' => array(
        'ZfDeals\Controller\Admin'
                => 'ZfDeals\Controller\AdminControllerFactory'
    )
)
// [..]
```

Listing 26.24

Unit tests for my controller

Not yet perfect, but a lot better. At least I can now add some unit tests for my controller. Before the refactoring, I would not have been able to properly unit test my controller at all.

But what exactly to test in a controller? In best case, a controller itself doesn't do a lot magic. Mainly a controller should instruct other objects to do the heavy lifting. What I should test here is that the controller instructs the right objects at the right time to do the right things. Mainly, we have the following cases our controller has to deal with:

1. The form is requested using `GET`: The web form must be shown.
2. The form is submitted using `POST` but validation failed: Show the form again with error messages.
3. The form is submitted using `POST` and validated successfully: A new entity has to be created based on the data given and stored in the database using the mapper.

And this is how the tests could look like:

```php
<?php
namespace ZfDeals\ControllerTest;

use ZfDeals\Controller\AdminController;
use Zend\Http\Request;
use Zend\Http\Response;
use Zend\Mvc\MvcEvent;
use Zend\Mvc\Router\RouteMatch;

class AdminControllerTest extends \PHPUnit_Framework_TestCase
{
        private $controller;
        private $request;
        private $response;
        private $routeMatch;
        private $event;

        public function setUp()
        {
                $this->controller = new AdminController();
                $this->request = new Request();
                $this->response = new Response();
                $this->routeMatch = new RouteMatch(array('controller' => 'admin'));
                $this->routeMatch->setParam('action', 'add-product');
                $this->event = new MvcEvent();
                $this->event->setRouteMatch($this->routeMatch);
                $this->controller->setEvent($this->event);
        }

        public function testShowFormOnGetRequest()
        {
                $fakeForm = new \Zend\Form\Form('fakeForm');
                $this->controller->setProductAddForm($fakeForm);
                $this->request->setMethod('get');
                $response = $this->controller->dispatch($this->request);
                $viewModelValues = $response->getVariables();
                $formReturned = $viewModelValues['form'];
                $this->assertEquals($formReturned->getName(), $fakeForm->getName());
}

        public function testShowFormOnValidationError()
        {
                $fakeForm = $this->getMock('Zend\Form\Form', array('isValid'));

                $fakeForm->expects($this->once())
                        ->method('isValid')
                        ->will($this->returnValue(false));

                $this->controller->setProductAddForm($fakeForm);
                $this->request->setMethod('post');
                $response = $this->controller->dispatch($this->request);
                $viewModelValues = $response->getVariables();
                $formReturned = $viewModelValues['form'];
                $this->assertEquals($formReturned->getName(), $fakeForm->getName());
        }

        public function testCallMapperOnFormValidationSuccess()
        {
                $fakeForm = $this->getMock(
                        'Zend\Form\Form', array('isValid', 'getData')
                );

                $fakeForm->expects($this->once())
                        ->method('isValid')
                        ->will($this->returnValue(true));

                $fakeForm->expects($this->once())
                        ->method('getData')
                        ->will($this->returnValue(new \stdClass()));

                $fakeMapper = $this->getMock('ZfDeals\Mapper\Product',
                        array('insert'),
                        array(),
                        '',
                        false
                );

                $fakeMapper->expects($this->once())
                        ->method('insert')
                        ->will($this->returnValue(true));

                $this->controller->setProductAddForm($fakeForm);
                $this->controller->setProductMapper($fakeMapper);
                $this->request->setMethod('post');
                $response = $this->controller->dispatch($this->request);
        }
}
```

Listing 26.25

A good starting point, I guess. To run all tests with a single command, I added them to `phpunit.xml` file:

```
<phpunit bootstrap="./bootstrap.php">
        <testsuites>
                <testsuite name="AllTests">
                        <directory>./ZfDealsTest/FormTest</directory>
                        <directory>./ZfDealsTest/ControllerTest</directory>
                </testsuite>
        </testsuites>
</phpunit>
```

Avoid ambiguous product IDs

There's one thing that comes to my mind: What actually happens, if I add the same product ID twice? I break the system as I configured the column to be unique in the database. It's the primary key. We can handle this situation by adding a proper try/catch statement:

```php
<?php
public function addProductAction()
{
    $form = $this->productAddForm;

    if ($this->getRequest()->isPost()) {
        $form->setData($this->getRequest()->getPost());

        if ($form->isValid()) {
            $model = new ViewModel(
                array(
                    'form' => $form
                )
            );

            try {
                $this->productMapper->insert($form->getData());
                $model->setVariable('success', true);
            } catch (\Exception $e) {
                $model->setVariable('insertError', true);
            }

            return $model;
        } else {
            return new ViewModel(
                array(
                    'form' => $form
                )
            );
        }
    } else {
        return new ViewModel(
            array(
                'form' => $form
            )
        );
    }
}
```

Listing 26.26

And the view file now looks like this:

```
<?php if ($this->success) { ?>
<div class="alert alert-success">Produkt hinzugefügt!</div>
<?php } ?>

<?php if ($this->insertError) { ?>
<div class="alert alert-error">
        Produkt konnte nicht hinzugefügt werden.
</div>
<?php } ?>

<?php
$this->form->prepare();
echo $this->form()->openTag($this->form);
echo $this->formRowTwb($this->form->get('product')->get('id'));
echo $this->formRowTwb($this->form->get('product')->get('name'));
echo $this->formRowTwb($this->form->get('product')->get('stock'));
echo $this->formSubmitTwb($this->form->get('submit'));
echo $this->form()->closeTag();
```

Listing 26.27

Let’s add another test to verify the try/catch statements works as desired. In the course of adding the test, I do some test code cleanup and extract the code to create fake objects:

```
<?php
namespace ZfDeals\ControllerTest;

use ZfDeals\Controller\AdminController;
use Zend\Http\Request;
use Zend\Http\Response;
use Zend\Mvc\MvcEvent;
use Zend\Mvc\Router\RouteMatch;

class AdminControllerTest extends \PHPUnit_Framework_TestCase
{
        private $controller;
        private $request;
        private $response;
        private $routeMatch;
        private $event;

        public function setUp()
        {
```

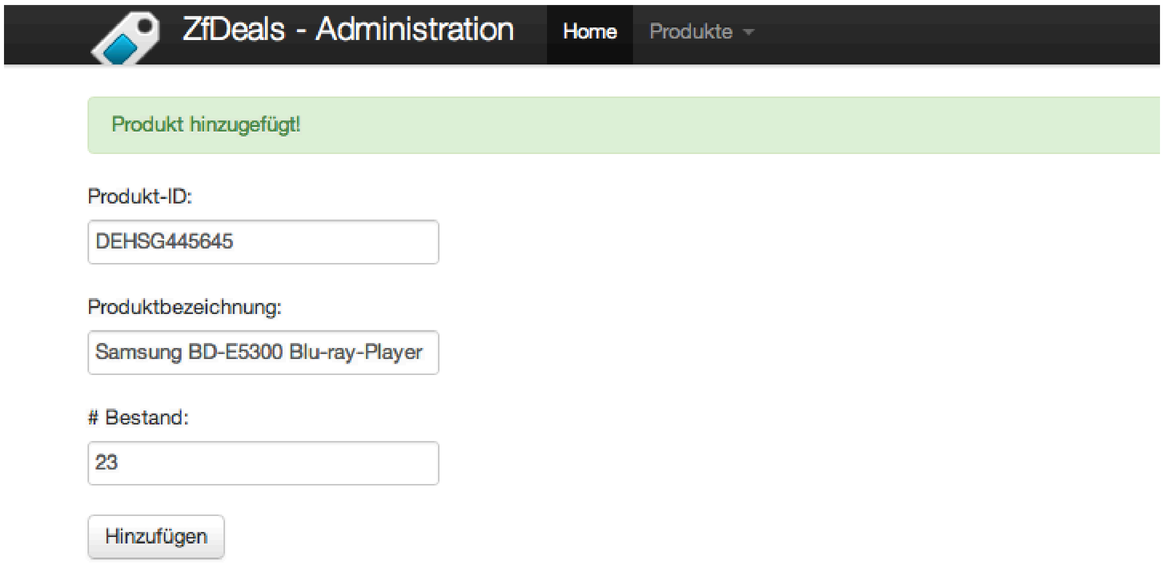
```
        $this->controller = new AdminController();
        $this->request = new Request();
        $this->response = new Response();
        $this->routeMatch = new RouteMatch(array('controller' => 'admin'));
        $this->routeMatch->setParam('action', 'add-product');
        $this->event = new MvcEvent();
        $this->event->setRouteMatch($this->routeMatch);
        $this->controller->setEvent($this->event);
    }

    public function testShowFormOnGetRequest()
    {
        $fakeForm = new \Zend\Form\Form('fakeForm');
        $this->controller->setProductAddForm($fakeForm);
        $this->request->setMethod('get');
        $response = $this->controller->dispatch($this->request);
        $viewModelValues = $response->getVariables();
        $formReturned = $viewModelValues['form'];
        $this->assertEquals($formReturned->getName(), $fakeForm->getName());
    }

    public function testShowFormOnValidationError()
    {
        $fakeForm = $this->getFakeForm(false);
        $this->controller->setProductAddForm($fakeForm);
        $this->request->setMethod('post');
        $response = $this->controller->dispatch($this->request);
        $viewModelValues = $response->getVariables();
        $formReturned = $viewModelValues['form'];
        $this->assertEquals($formReturned->getName(), $fakeForm->getName());
    }

    public function testCallMapperOnFormValidationSuccessPersistenceSuccess()
    {
        $fakeForm = $this->getFakeForm();

        $fakeForm->expects($this->once())
                ->method('getData')
                ->will($this->returnValue(new \stdClass()));

        $fakeMapper = $this->getFakeMapper();

```

```
                $fakeMapper->expects($this->once())
                        ->method('insert')
                        ->will($this->returnValue(true));

                $this->controller->setProductAddForm($fakeForm);
                $this->controller->setProductMapper($fakeMapper);
                $this->request->setMethod('post');
                $response = $this->controller->dispatch($this->request);
                $viewModelValues = $response->getVariables();
                $this->assertTrue(isset($viewModelValues['success']));
        }

        public function testCallMapperOnFormValidationSuccessPersistenceError()
        {
                $fakeForm = $this->getFakeForm();

                $fakeForm->expects($this->once())
                        ->method('getData')
                        ->will($this->returnValue(new \stdClass()));

                $fakeMapper = $this->getFakeMapper();

                $fakeMapper->expects($this->once())
                        ->method('insert')
                        ->will($this->throwException(new \Exception));

                $this->controller->setProductAddForm($fakeForm);
                $this->controller->setProductMapper($fakeMapper);
                $this->request->setMethod('post');
                $response = $this->controller->dispatch($this->request);
                $viewModelValues = $response->getVariables();
                $this->assertTrue(isset($viewModelValues['form']));
                $this->assertTrue(isset($viewModelValues['insertError']));
        }

        public function getFakeForm($isValid = true)
        {
                $fakeForm = $this->getMock(
                        'Zend\Form\Form', array('isValid', 'getData')
                );

                $fakeForm->expects($this->once())
```

```
                    ->method('isValid')
                    ->will($this->returnValue($isValid));

            return $fakeForm;
        }

        public function getFakeMapper()
        {
            $fakeMapper = $this->getMock('ZfDeals\Mapper\Product',
                array('insert'),
                array(),
                '',
                false
            );

            return $fakeMapper;
        }
}
```

Listing 26.28

And this is how it looks like now:



Produkt hinzugefügt!

Produkt-ID: DEHSG445645

Produktbezeichnung: Samsung BD-E5300 Blu-ray-Player

Bestand: 23

ZfDeals - Successfully added a new product

Congrats - that was it for Sprint 2!

Sprint 3 - Add a deal, show available deals

Source Code Download

The source code of Sprint 3 can be downloaded using Tag "Sprint3" on GitHub[a].

[a]https://github.com/michael-romer/ZfDealsApp/tree/Sprint3

The first user story of Sprint 3 reads as follows:

> "In order to offer a deal as a merchant, I want add one to the system."

Acceptance criteria:

- By using a web form, a new deal with pricing details, start date, end date and stock information can be added based on an existing product within the system

The second user story for this sprint reads as follows:

> "In order to buy a product as a customer, I want to see all available deals at a glance."

Acceptance criteria:

- All deals are displayed that are currently available.
- Only deals are displayed that have stock.

But before I start working on the new requirements, I will work on some technical improvements.

Coding Standard

ZF2 complies to the PSR-2 Coding Standard[87]. Right from the beginning, I want to follow the PSR-2 rules as well so that I can be for sure that my code will look familiar to others that needs to understand, modify or enhance it. To support PSR-2 adoption, one may install PHP_Codesniffer[88]. It's pre-installed on my local box, however, it may be installed quickly using PEAR[89]:

[87]https://github.com/pmjones/fig-standards/blob/psr-1-style-guide/proposed/PSR-2-advanced.md
[88]http://pear.php.net/package/PHP_CodeSniffer/download/
[89]http://pear.php.net/

```
$ pear install PHP_CodeSniffer-1.3.6
```

Now, when running

```
$ phpcs -v --standard=psr2 ZfDeals/
```

in directory `modules`, PHP_Codesniffer automatically tests my `ZfDeals` module code on PSR-2 compliance. Unfortunately, with the current code, PHP_Codesniffer has a lot to criticise:

```
FILE: /vagrant/module/ZfDeals/Module.php
----------------------------------------------------------------------
FOUND 7 ERROR(S) AFFECTING 3 LINE(S)
----------------------------------------------------------------------
11 | ERROR | Opening parenthesis of a multi-line
function call must be the last content on the line

11 | ERROR | Only one argument is allowed per line
in a multi-line function call

11 | ERROR | Only one argument is allowed per line
in a multi-line function call

11 | ERROR | Expected 1 space after FUNCTION keyword; 0 found

14 | ERROR | Only one argument is allowed per line
in a multi-line function call

14 | ERROR | Closing parenthesis of a multi-line function
call must be on a line by itself

32 | ERROR | Expected 1 blank line at end of file; 0 found
```

Once all CS issues are fixed, PHP_Codesniffer approves my code in silence and no output is given to the command line.

Fix CS issues automatically

Fabien Potencier recently released a very helpful tool called "PHP-CS-Fixer" to automatically fix most of the typical CS issues. It's worth taking a look.

To make my coding style checks as easy as possible, I create a little helper script in `tests`:

```php
<?php
echo shell_exec('phpcs --standard=psr2 ../module/ZfDeals') . PHP_EOL;
```

Listing 26.29

I can now easily check my code on PSR-2 compliance by executing:

```
$ php checkstyle.php
```

Database init script

`ZfDeals` requires a specific database schema to work. A so called DDL script in `/module/ZfDeals/data` makes it easy to create the schema needed:

```sql
CREATE TABLE product(
        id varchar(255) NOT NULL,
        name varchar(255) NOT NULL,
        stock int(10) NOT NULL, PRIMARY KEY (id)
);
```

It's a starting point. More data definitions will surely be added soon.

Deal entity

Back to this Sprint's user stories. First, I code the `Deal` entity class:

```php
<?php
namespace ZfDeals\Entity;

class Deal
{
        protected $id;
        protected $price;
        protected $startDate;
        protected $endDate;
        protected $product;

        public function setEndDate($endDate)
        {
                $this->endDate = $endDate;
        }

        public function getEndDate()
        {
                return $this->endDate;
        }

        public function setStartDate($startDate)
        {
                $this->startDate = $startDate;
        }

        public function getStartDate()
        {
                return $this->startDate;
        }

        public function setId($id)
        {
                $this->id = $id;
        }

        public function getId()
        {
                return $this->id;
        }

        public function setPrice($price)
        {
                $this->price = $price;
        }

        public function getPrice()
        {
                return $this->price;
        }

        public function setProduct($product)
        {
                $this->product = $product;
        }

        public function getProduct()
        {
                return $this->product;
        }
}
```

Listing 26.30

A deal has a price, start date, end date and a reference to the product on sale. The following data structure is needed for persistence of deals and therefore I add it to `structure.sql`:

```
CREATE TABLE deal(
        id int(10) NOT NULL AUTO_INCREMENT,
        price float NOT NULL,
        startDate date NOT NULL,
        endDate date NOT NULL,
        product varchar(255) NOT NULL,
        PRIMARY KEY (id)
);
```

Add a new deal

I add a section "Deals" with entry "Add Deal" to the admin's navigation panel. It points to `/deals/admin/deal/add`. As usual, to make things work, I set up a route in `module.config.php`. The `DealAdd` form is used to add a new deal to the system:

```
<?php
namespace ZfDeals\Form;

use Zend\Form\Form;
use Zend\ServiceManager\ServiceManager;
use Zend\ServiceManager\ServiceManagerAwareInterface;

class DealAdd extends Form
{
        public function __construct()
        {
                parent::__construct('dealAdd');
                $this->setAttribute('action', '/deals/admin/deal/add');
                $this->setAttribute('method', 'post');

                $this->add(
                        array(
                                'type' => 'ZfDeals\Form\DealFieldset',
                                'options' => array(
                                        'use_as_base_fieldset' => true
                                )
                        )
                );

                $this->add(
                        array(
                                'name' => 'submit',
                                'attributes' => array(
                                        'type'  => 'submit',
                                        'value' => 'Hinzufügen'
                                ),
                        )
                );
        }
}
```

Listing 26.31

`DealFieldset` definitions:

```
<?php
namespace ZfDeals\Form;

use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterInterface;

class DealFieldset extends Fieldset
{
        public function __construct()
        {
                parent::__construct('deal');

                $this->add(
                        array(
                                'name' => 'product',
                                'type' => 'ZfDeals\Form\ProductSelectorFieldset',
                        )
                );

                $this->add(
                        array(
                                'name' => 'price',
                                'type' => 'Zend\Form\Element\Number',
                                'attributes' => array(
                                        'step' => 'any'
                                ),
                                'options' => array(
                                        'label' => 'Preis:',
                                )
                        )
                );

                $this->add(
                        array(
                                'name' => 'startDate',
                                'type' => 'Zend\Form\Element\Date',
                                'options' => array(
                                        'label' => 'Startdatum:'
                                ),
                        )
                );

                $this->add(
                        array(
                                'name' => 'endDate',
                                'type' => 'Zend\Form\Element\Date',
                                'options' => array(
                                        'label' => 'Enddatum:'
                                ),
                        )
                );
        }
}
```

Listing 26.32

`DealFieldset` contains `ProductSelectorFieldset` that serves as a product chooser while adding a new deal:

```php
<?php
namespace ZfDeals\Form;

use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterInterface;

class ProductSelectorFieldset extends Fieldset
{
        public function __construct()
        {
                parent::__construct('productSelector');
                $this->setHydrator(new\Zend\Stdlib\Hydrator\Reflection());
                $this->setObject(new \ZfDeals\Entity\Product());

                $this->add(
                        array(
                                'name' => 'id',
                                'type'  => 'Zend\Form\Element\Select',
                                'options' => array(
                                        'label' => 'Produkt-ID:',
                                        'value_options' => array(
                                                '1' => 'Label 1',
                                                '2' => 'Label 2',
                                        ),
                                )
                        )
                );
        }
}
```

Listing 26.33

The list is initialized with dummy data only. Real products are added by the controller before actually rendering the form.

INTL extension

Opening the new URL my result in a "PHP Fatal error: Class 'NumberFormatter' not found". The reason is that PHP's INTL extension might be missing on the system but is needed by the framework, even if you don't directly deal with ZF2's I18N features. On a Linux system, one can easily install the INTL extension executing `apt-get install php5-intl`. Don't forget to restart Apache afterwards.

Form processing is done in `AdminController`. I extend `AdminControllerFactory` in a way, that `DealAdd` form is injected as well:

```php
<?php
namespace ZfDeals\Controller;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class AdminControllerFactory implements FactoryInterface
{
	public function createService(ServiceLocatorInterface $serviceLocator)
	{
		$ctr = new AdminController();
		$productAddForm = new \ZfDeals\Form\ProductAdd();
		$productAddForm->setHydrator(new\Zend\Stdlib\Hydrator\Reflection());
		$productAddForm->bind(new \ZfDeals\Entity\Product());
		$ctr->setProductAddForm($productAddForm);

		$mapper = $serviceLocator->getServiceLocator()
			->get('ZfDeals\Mapper\Product');

		$ctr->setProductMapper($mapper);
		$dealAddForm = new \ZfDeals\Form\DealAdd();
		$ctr->setDealAddForm($dealAddForm);

		$dealAddForm
			->setHydrator(new\Zend\Stdlib\Hydrator\Reflection());

		$dealAddForm->bind(new \ZfDeals\Entity\Deal());

		$dealMapper = $serviceLocator->getServiceLocator()
			->get('ZfDeals\Mapper\Deal');

		$ctr->setDealMapper($dealMapper);
		return $ctr;
	}
}
```

Listing 26.34

Form processing code is located in `addDealAction` in `AdminController`. First, I make sure `ProductSelectorFieldset` is initialized with real product data retrieved from the database. `DealMapper` takes care of adding deals to the database as well as finding active deals:

```php
<?php

namespace ZfDeals\Mapper;

use ZfDeals\Entity\Deal as DealEntity;
use Zend\Stdlib\Hydrator\HydratorInterface;
use Zend\Db\TableGateway\TableGateway;
use Zend\Db\TableGateway\Feature\RowGatewayFeature;
use Zend\Db\Sql\Sql;
use Zend\Db\Sql\Insert;

class Deal extends TableGateway
{
        protected $tableName  = 'deal';
        protected $idCol = 'id';
        protected $entityPrototype = null;
        protected $hydrator = null;

        public function __construct($adapter)
        {
                parent::__construct(
                        $this->tableName,
                        $adapter,
                        new RowGatewayFeature($this->idCol)
                );

                $this->entityPrototype = new DealEntity();
                $this->hydrator = new \Zend\Stdlib\Hydrator\Reflection;
        }

        public function insert($entity)
        {
                return parent::insert($this->hydrator->extract($entity));
        }

        public function findActiveDeals()
        {
                $sql = new \Zend\Db\Sql\Sql($this->getAdapter());
                $select = $sql->select()
                        ->from($this->tableName)
                        ->join('product', 'deal.product=product.id')
                        ->where('DATE(startDate) <= DATE(NOW())')
                        ->where('DATE(endDate) >= DATE(NOW())')
                        ->where('stock > 0');

                $stmt = $sql->prepareStatementForSqlObject($select);
                $results = $stmt->execute();

                return $this->hydrate($results);
        }

        public function hydrate($results)
        {
                $deals = new \Zend\Db\ResultSet\HydratingResultSet(
                        $this->hydrator,
                        $this->entityPrototype
                );

                return $deals->initialize($results);
        }
}
```

Listing 26.35

My `addDealAction` still is straightforward, however, things already start to get a bit messy:

```
<?php
// [..]
public function addDealAction()
{
    $form = $this->dealAddForm;

    $products = $this->productMapper->select();
    $fieldElements = array();

    foreach ($products as $product) {
        $fieldElements[$product['id']] = $product['name'];
    }

    $form->get('deal')->get('product')
```

```
            ->get('id')->setValueOptions($fieldElements);

    if ($this->getRequest()->isPost()) {
        $form->setData($this->getRequest()->getPost());

        if ($form->isValid()) {
            $model = new ViewModel(
                array(
                    'form' => $form
                )
            );

            $newDeal = $form->getData();
            $newDeal->setProduct($newDeal->getProduct()->getId());

            try {
                $this->dealMapper->insert($newDeal);
                $model->setVariable('success', true);
            } catch (\Exception $e) {
                $model->setVariable('insertError', true);
            }

            return $model;
        } else {
            return new ViewModel(
                array(
                    'form' => $form
                )
            );
        }
    } else {
        return new ViewModel(
            array(
                'form' => $form
            )
        );
    }
}
// [..]
```

Listing 26.36

Show active deals

I create `IndexController` with its factory `IndexControllerFactory` to show active deals to customers. `IndexControllerFactory` injects the `Deal` mapper to retrieve active deals from the database:

```php
<?php
namespace ZfDeals\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Zend\Form\Annotation\AnnotationBuilder;

class IndexController extends AbstractActionController
{
        private $dealMapper;
        private $productMapper;

        public function indexAction()
        {
                $deals = $this->dealMapper->findActiveDeals();
                $dealsView = array();

                foreach ($deals as $deal) {
                        $deal->setProduct(
                                $this->productMapper->findOneById($deal->getProduct())
                        );

                        $dealsView[] = $deal;
                }

                return new ViewModel(
                        array(
                                'deals' => $dealsView
                        )
                );
        }

        public function setDealMapper($dealMapper)
        {
                $this->dealMapper = $dealMapper;
        }

        public function getDealMapper()
        {
                return $this->dealMapper;
        }

        public function setProductMapper($productMapper)
        {
                $this->productMapper = $productMapper;
        }

        public function getProductMapper()
        {
                return $this->productMapper;
        }
}
```

Listing 26.37

The `foreach` isn't very elegant here, but as I don't except high volume usage for now, I guess I can live with it for the moment and improve it later. However, I know it's a performance bottleneck.

I will use another layout template for showing deals to customers. It's configured in class `Module` within `init`:

```
<?php
[..]
public function init(ModuleManager $moduleManager)
{
    $sharedEvents = $moduleManager->getEventManager()->getSharedManager();

    $sharedEvents->attach(
        'ZfDeals\Controller\AdminController',
        'dispatch',
        function ($e) {
            $controller = $e->getTarget();
            $controller->layout('zf-deals/layout/admin');
        },
        100
    );

    $sharedEvents->attach(
        'ZfDeals\Controller\IndexController',
        'dispatch',
        function ($e) {
            $controller = $e->getTarget();
            $controller->layout('zf-deals/layout/site');
        },
        100
    );
}
[..]
```

Listing 26.38

Now `/deals` brings up all active deals available in the system.

A custom controller type for dealing with forms

I am not really happy with the code yet. First, let's take a look at `AdminController`: It holds both "add actions". One for products, one for deals. Both methods look a lot like "copy and paste" as they mainly do the same thing: handling a web form. And both look somewhat complicated and a bit too nested. The reason is that a single action displays the form, does data validation, error handling as well as data persistence. Maybe that's too much responsibility for a single action and it should be refactored. The same is true for its template file. Another issue is that `IndexControllerFactory` by defaults creates and injects both forms everytime, even if only one may be used at a time. This doesn't look appropriate.

To fix all issues mentioned at one dash, I take advantage of the fact, that a Controller in ZF2 simply requires to have a method `onDispatch()` to be fully functional, creating a `Response` based on a `Request` given. This allows me to come up with a special type of Controller built for dealing with forms in its very own way, an abstract controller class called `AbstractFormController`:

```
<?php
namespace ZfDeals\Controller;

use Zend\Mvc\Controller\AbstractController;
use Zend\Mvc\MvcEvent;
use Zend\Form\Form as Form;
use Zend\View\Model\ViewModel;

abstract class AbstractFormController extends AbstractController
{
        protected $form;

        public function __construct(Form $form)
        {
                $this->form = $form;
        }

        public function onDispatch(MvcEvent $e)
        {
                if (method_exists($this, 'prepare')) {
                        $this->prepare();
                }

                $routeMatch = $e->getRouteMatch();

                if ($this->getRequest()->isPost()) {
                        $this->form->setData($this->getRequest()->getPost());

                        if ($this->form->isValid()) {
                                $routeMatch->setParam('action', 'process');
                                $return = $this->process();
                        } else {
                                $routeMatch->setParam('action', 'error');
                                $return = $this->error();
                        }

                } else {
                        $routeMatch->setParam('action', 'show');
                        $return = $this->show();
                }

                $e->setResult($return);
                return $return;
        }

        abstract protected function process();

        protected function show()
        {
                return new ViewModel(
                        array(
                                'form' => $this->form
                        )
                );
        }

        protected function error()
        {
                return new ViewModel(
                        array(
                                'form' => $this->form
                        )
                );
        }

        public function setForm($form)
        {
                $this->form = $form;
        }

        public function getForm()
        {
                return $this->form;
        }
}
```

Listing 26.39

And this is how it works: At routing a `AbstractFormController` based controller is matched to the URL requested. Its `dispatch()` and then its `onDispatch()` method is called. Instead of calling an action method, which usually happens when the framework's well-known `AbstractActionController` is used, another method is called based on the state of form processing. Let me explain that further.

If the url requested is using HTTP's GET method, it's obvious that we simply have to initially display the form to the user. Therefore, `show()` is called. It's already given by `AbstractFormController`. A concrete controller based on `AbstractFormController` may overwrite this method if it has special needs displaying the form. `AbstractFormController` is designed in a way, that on instantiation, the form in question has to be injected. This way, the controller can simply call the form's `isValid()` method on a `POST` request. If validation was successful, `process()` is executed. If not, method `error()` is called instead. As processing a successfully validated form usually is specific to the form in question, `AbstractFormController` only ships with an abstract method `process()`. It needs to be implemented individually by the application developer.

In short, `AbstractFormController` helps to prevent "Spaghetti code[90]" in controllers dealing with forms. You may wonder why `setParam()` still is used to set values for key `action`. It's simply to ensure that template lookup later is still working. Therefore, `AbstractFormController` needs a `show.phtml`, `process.phtml` and `error.phtml` template to work properly.

Now, with `AbstractFormController` in place, each form gets its own controller. `ProductAddFormController` looks like this:

[90] http://en.wikipedia.org/wiki/Spaghetti_code

```php
<?php
namespace ZfDeals\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Zend\Stdlib\Hydrator\Reflection;
use ZfDeals\Entity\Product as ProductEntity;
use ZfDeals\Form\ProductAdd as ProductAddForm;

class ProductAddFormController extends AbstractFormController
{
        private $productMapper;

        public function __construct(ProductAddForm $form)
        {
                parent::__construct($form);
        }

        public function prepare()
        {
                $this->form->setHydrator(new Reflection());
                $this->form->bind(new ProductEntity());
        }

        public function process()
        {
                $model = new ViewModel(
                        array(
                                'form' => $this->form
                        )
                );

                try {
                        $this->productMapper->insert($this->form->getData());
                        $model->setVariable('success', true);
                } catch (\Exception $e) {
                        $model->setVariable('insertError', true);
                }

                return $model;
        }

        public function setProductMapper($productMapper)
        {
                $this->productMapper = $productMapper;
        }

        public function getProductMapper()
        {
                return $this->productMapper;
        }
}
```

Listing 26.40

`DealAddFormController` like this:

```
<?php
namespace ZfDeals\Controller;

use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;
use Zend\Stdlib\Hydrator\Reflection;
use ZfDeals\Entity\Deal as DealEntity;

class DealAddFormController extends AbstractFormController
{
        private $productMapper;
        private $dealMapper;

        public function prepare()
        {
                $this->form->setHydrator(new Reflection());
                $this->form->bind(new DealEntity());

                $products = $this->productMapper->select();
                $fieldElements = array();

                foreach ($products as $product) {
                        $fieldElements[$product['id']] = $product['name'];
                }

                $this->form->get('deal')
                        ->get('product')
                        ->get('id')->setValueOptions($fieldElements);
        }

        public function process()
        {
                $model = new ViewModel(
                        array(
                                'form' => $this->form
                        )
                );

                $newDeal = $this->form->getData();
                $newDeal->setProduct($newDeal->getProduct()->getId());

                try {
                        $this->dealMapper->insert($newDeal);
                        $model->setVariable('success', true);
                } catch (\Exception $e) {
                        $model->setVariable('insertError', true);
                }

                return $model;
        }

        public function setProductMapper($productMapper)
        {
                $this->productMapper = $productMapper;
        }

        public function getProductMapper()
        {
                return $this->productMapper;
        }

        public function setDealMapper($dealMapper)
        {
                $this->dealMapper = $dealMapper;
        }

        public function getDealMapper()
        {
                return $this->dealMapper;
        }
```

```
}
```

Listing 26.41

Method `prepare()` allows for initialization code run before form processing, if implemented by a controller.

Unit tests for AbstractFormController

Now I add unit tests for `AbstractFormController`. This is where I will test all general form processing logic in a single place, saving me a lot of time as I can now ditch most of the individual form tests:

```php
<?php
namespace ZfDeals\ControllerTest;

use ZfDeals\Controller\AbstractFormController;
use Zend\Http\Request;
use Zend\Http\Response;
use Zend\Mvc\MvcEvent;
use Zend\Mvc\Router\RouteMatch;
use ZfDeals\Form\ProductAdd as ProductAddForm;

class AbstractFormControllerTest extends \PHPUnit_Framework_TestCase
{
        private $controller;
        private $request;
        private $response;
        private $routeMatch;
        private $event;

        public function setUp()
        {
                $fakeController = $this->getMockForAbstractClass(
                        'ZfDeals\Controller\AbstractFormController',
                        array(),
                        '',
                        false
                );

                $this->controller = $fakeController;
                $this->request = new Request();
                $this->response = new Response();

                $this->routeMatch = new RouteMatch(
                        array('controller' => 'abstract-form')
                );

                $this->event = new MvcEvent();
                $this->event->setRouteMatch($this->routeMatch);
                $this->controller->setEvent($this->event);
        }

        public function testShowOnGetRequest()
        {
                $this->form = new \Zend\Form\Form('fakeForm');
                $this->controller->setForm($this->form);
                $this->request->setMethod('get');
                $response = $this->controller->dispatch($this->request);
                $viewModelValues = $response->getVariables();
                $formReturned = $viewModelValues['form'];

                $this->assertEquals(
                    $formReturned->getName(), $this->form->getName()
                );
        }

        public function testErrorOnValidationError()
        {
                $fakeForm = $this->getMock(
                        'Zend\Form\Form', array('isValid')
                );

                $fakeForm->expects($this->once())
                        ->method('isValid')
                        ->will($this->returnValue(false));

                $this->controller->setForm($fakeForm);
                $this->request->setMethod('post');
                $response = $this->controller->dispatch($this->request);
                $viewModelValues = $response->getVariables();
                $formReturned = $viewModelValues['form'];
                $this->assertEquals($formReturned, $fakeForm);
        }
```

```

	public function testProcessOnValidationSuccess()
	{
	    $fakeForm = $this->getMock(
	            'Zend\Form\Form', array('isValid')
	    );

		$fakeForm->expects($this->once())
			->method('isValid')
			->will($this->returnValue(true));

		$this->controller->setForm($fakeForm);
		$this->request->setMethod('post');

		$this->controller->expects($this->once())
			->method('process')
			->will($this->returnValue(true));

		$response = $this->controller->dispatch($this->request);
	}
}
```

Listing 26.42

Use closures as simple factories

In my code, I have multiple full-blown factories only doing tiny work. `DealAddFormControllerFactory` for instance only retrieves some services from `ServiceManager` and injects them on controller creation:

```
<?php
namespace ZfDeals\Controller;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class DealAddFormControllerFactory implements FactoryInterface
{
	public function createService(ServiceLocatorInterface $serviceLocator)
	{
		$form = new \ZfDeals\Form\DealAdd();
		$ctr = new DealAddFormController($form);
```

```php

                $dealMapper = $serviceLocator
                        ->getServiceLocator()->get('ZfDeals\Mapper\Deal');

                $ctr->setDealMapper($dealMapper);

                $productMapper = $serviceLocator->getServiceLocator()
                        ->get('ZfDeals\Mapper\Product');

                $ctr->setProductMapper($productMapper);
                return $ctr;
        }
}
```

Listing 26.43

To save lines of code (and though lower the risk of bugs in my application) I migrate the factories to simple closures in module's config file:

```php
<?php
// [..]
'controllers' => array(
    'invokables' => array(
        'ZfDeals\Controller\Admin'
                => 'ZfDeals\Controller\AdminController',
    ),
    'factories' => array(
        'ZfDeals\Controller\DealAddForm' => function ($serviceLocator) {
            $form = new ZfDeals\Form\DealAdd();
            $ctr = new ZfDeals\Controller\DealAddFormController($form);

            $dealMapper = $serviceLocator
                    ->getServiceLocator()->get('ZfDeals\Mapper\Deal');

            $ctr->setDealMapper($dealMapper);

            $productMapper = $serviceLocator->getServiceLocator()
                    ->get('ZfDeals\Mapper\Product');

            $ctr->setProductMapper($productMapper);
            return $ctr;
        },
        'ZfDeals\Controller\ProductAddForm' => function ($serviceLocator) {
            $form = new \ZfDeals\Form\ProductAdd();
            $ctr = new ZfDeals\Controller\ProductAddFormController($form);

            $productMapper = $serviceLocator->getServiceLocator()
                    ->get('ZfDeals\Mapper\Product');

            $ctr->setProductMapper($productMapper);
            return $ctr;
        },
        'ZfDeals\Controller\Index' => function ($serviceLocator) {
            $ctr = new ZfDeals\Controller\IndexController();

            $productMapper = $serviceLocator->getServiceLocator()
                    ->get('ZfDeals\Mapper\Product');

            $dealMapper = $serviceLocator->getServiceLocator()
                    ->get('ZfDeals\Mapper\Deal');

            $ctr->setDealMapper($dealMapper);
            $ctr->setProductMapper($productMapper);
            return $ctr;
        }
    ),
),
```

Listing 26.44

And that was it for Sprint 3!

Sprint 4 - Order form

Source Code Download

The source code of Sprint 4 can be downloaded using Tag "Sprint4" on GitHub[a].

[a] https://github.com/michael-romer/ZfDealsApp/tree/Sprint4

Again, before working on this Sprint's requirements, I use the break between the last and the current sprint, called "slack time", to further get my code base in a better shape. Today, I first want to know what my "code coverage" is. It describes the degree to which the source code of a program has been tested. Thanks to PHPUnit, this is an easy task. I simple need to add section `logging` to `phpunit.xml`:

```
<phpunit bootstrap="./bootstrap.php">
        <testsuites>
                <testsuite name="AllTests">
                        <directory>./ZfDealsTest/FormTest</directory>
                        <directory>./ZfDealsTest/ControllerTest</directory>
                </testsuite>
        </testsuites>
        <logging>
                <log type="coverage-html"
                target="./reports/coverage"
                charset="UTF-8"
                yui="true"
                highlight="false"
                lowUpperBound="35"
                highLowerBound="70" />
        </logging>
</phpunit>
```

Now when executing

```
$ phpunit
```

not only all test are executed, but now also "Code Coverage Reports" in HTML format are written to disk. They show the total values of coverage at a glance as well as detailed information on single lines of code:

```
ZfDeals: 55.16%
        config: 0%
        src: 67.51%
                Controller: 90.41%
                Entity: 25%
                Form: 79.31%
                Mapper: 0%
        Module.php: 0%
```

What I see is obviously not ideal. In general, a code coverage of 70-80% is desirable and I see that in some areas I don't have any tests at all. I will need to work that that for sure.

There's one other thing before I get to the User Stories of Sprint 4: I want to make sure, right from the beginning, that ZfDeals' user interface supports multiple languages and ships at least with English and German translations. First, I remove all language files shipped with `ZendSkeletonApplication` in module `Application`. I don't need them. I also drop my custom form validation error messages and use the ZF2 default error messages which are already translated into the most common languages. Therefore, I copy `Zend_Validate.php` from `vendor/zendframework/zendframework/resources/languages` to directory `language` of ZfDeals and configure `translator` in `module.config.php` to pick up this language file:

```php
<?php
// [..]
'translator' => array(
    'locale' => 'de_DE',
    'translation_file_patterns' => array(
        array(
            'type'     => 'PhpArray',
            'base_dir' => __DIR__ . '/../language',
            'pattern'  => '%s.php',
        ),
    )
)
// [..]
```

Listing 26.45

In addition, I configure my validators to apply `translator` to their error messages by default:

```php
<?php
// [..]
public function onBootstrap($e)
{
    \Zend\Validator\AbstractValidator::setDefaultTranslator(
        $e->getApplication()->getServiceManager()->get('translator')
    );

    $eventManager        = $e->getApplication()->getEventManager();
    $moduleRouteListener = new ModuleRouteListener();
    $moduleRouteListener->attach($eventManager);
}
// [..]
```

Listing 26.46

Now I once run through all view templates and forms of the module to make sure `translator` is used whenever text is displayed.

The User Story of this sprint reads as follows:

> "In order to buy a product with a special discount price as a customer, I want to fill the order form."

Acceptance criteria:

- All active deals shown on the site have a "Buy" button.
- Clicking the "Buy" button brings the customer to a web form prompting for customer name and shipping address. All data is mandatory.
- A list of all orders received is added to the admin section of ZfDeals.

This more or less now already looks like business as usual. For the order form, I set up a route, a form, a controller and entity `order`. `Order` keeps a reference to a `Product` and stores all additional order data. A new mapper for `Order` takes care of order persistence. Service `Checkout` is ZfDeals's first "Business Service". `Checkout` takes care of adding a new `Order` to the system as well as lowering the stock of a product ordered by one. The reason I put this code into a dedicated service is because I touch two different entities so it doesn't naturally fit into one of them. In addition, I don't want to put the checkout logic into `CheckoutFormController` because I maybe want to re-use this logic elsewhere, in another Controller or web service. Mainly, the `CheckoutService` consists of a method called `process()`:

```php
<?php
public function process($ordering)
{
        try {
                $this->orderMapper->insert($ordering);
                $deal = $this->dealMapper->findOneById($ordering->getDeal());
                $product = $this->productMapper->findOneById($deal->getProduct());

                $this->productMapper->update(
                        array('stock' => $product->getStock() - 1),
                        array('productId' => $product->getProductId())
                );

        } catch (\Exception $e) {
                throw new \DomainException('Order could not be processed.');
        }

        return true;
}
```

Listing 26.47

The code is straight-forward. It surely does not yet handle any type of exception that may occur, but that's fine for now. At least, it does what it should.

And that's it!

Sprint 5 - Make ZfDeals available as a ZF2 module

The User Story of Sprint 5 reads as follows:

> "In order to have ZfDeals functionality in my own application as a developer, I want to add the ZfDeals using Composer to my own code."

A new repository for ZfDeals module

First of all, I set up a new git repository dedicated to the module's code. I move all module code from the host application to the new repository. Then I add the new repository (module code) as a git submodule to the existing one (host application code):

```
$ git submodule add https://github.com/michael-romer/ZfDeals module/ZfDeals
```

The command must be executed in the application root directory and it requires that the existing directory `ZfDeals` has been deleted before.

Now I have a handy development environment in place: In my workspace I have both, an application hosting my module as well as the module itself. As both are managed in their own repository, I can commit to them individually, based on where I run git commands on the shell. Also I have my module's source code separated from the rest so I can distribute it easily. In general, this is a good way to develop ZF2 modules whenever you want to keep a module's code separated.

While moving the files from one repository to the other, I include the module's static assets in `public` as well as the language file and unit tests. They all belong to the module code and shall be distributed along with it. Additionally, I move the database adapter out of ZfDeals' `module.config.php` and add it to the application's config instead. This means that if ZfDeals is used in an application, it requires the application to provide a database adapter to be used by ZfDeals. The same is true for `translator`. This is a common pattern for ZF2 modules: The most basic, system-wide services, such as a database adapter, shall be given by the host application and then be shared between all modules installed.

To make ZfDeals available through Composer, I add a `composer.json` file to the module:

```
{
        "name": "zf2book/zf-deals",
        "description": "This is the companion to the
                book 'Webentwicklung mit Zend Framework 2'",
        "type": "library",
        "keywords": [
                "zfdeals"
        ],
        "homepage": "http://zendframework2.de",
        "authors": [
                {
                        "name": "Michael Romer",
                        "email": "zf2buch@michael-romer.de",
                        "homepage": "http://zendframework2.de"
                }
        ],
        "require": {
                "php": ">=5.3.3",
                "zendframework/zendframework": "2.*"
        },
        "autoload": {
                "psr-0": {
                        "ZfDeals": "src/"
                },
                "classmap": [
                        "./Module.php"
                ]
        }
}
```

In section `autoload` all classes of directory `src` are configured to be autoloaded as well as the module's main `Module` class.

Now I add the module as a package to Packagist[91] using the identifier `zf2book/zf-deals`. After that, ZfDeal now can easily be installed using Composer. Last but not least I add `README.md` containing the installation instructions:

[91] https://packagist.org/packages/zf2book/zf-deals

```
Install
=======

Main Install
------------

1. Add the following statement to the requirements-block
of your composer.json: "zf2book/zf-deals": "dev-master",
"dlu/dlutwbootstrap": "dev-master"

2. Run a composer update to download the libraries needed.

3. Add "ZfDeals" and "DluTwBootstrap" to the list of active
modules in `application.config.php`

4. Import the SQL schema located in
`/vendor/zf2book/zf-deals/data/structure.sql`

5. Copy `/vendor/zf2book/zf-deals/data/public/zf-deals`
to the public folder of your application.

Post Install
------------

1. If you do not already have a valid
Zend\Db\Adapter\Adapter in your service manager configuration,
put the following in `/config/autoload/db.local.php`:

        <?php

        $dbParams = array(
            'database'  => 'changeme',
            'username'  => 'changeme',
            'password'  => 'changeme',
            'hostname'  => 'changeme',
        );

        return array(
            'service_manager' => array(
                'factories' => array(
                    'Zend\Db\Adapter\Adapter' => function
                            ($sm) use ($dbParams) {
```

```
                        return new Zend\Db\Adapter\Adapter(array(
                            'driver'    => 'pdo',
                            'dsn'       =>
'mysql:dbname='.$dbParams['database'].';host='.$dbParams['hostname'],
                            'database'  => $dbParams['database'],
                            'username'  => $dbParams['username'],
                            'password'  => $dbParams['password'],
                            'hostname'  => $dbParams['hostname'],
                        ));
                    },
                ),
            ),
        );

2. Navigate to http://yourproject/deals or http://yourproject/deals/admin
```

As one can see, besides installing the module via Composer, there is some more stuff to be done by a developer using ZfDeals:

- In addition to `ZfDeals` module `DluTwBootstrap` must be added to `application.config.php`. This is another ZF2 module we utilize for displaying our forms.
- The database structure given with `structure.sql` must be created.
- Static assets used by ZfDeals must be copied over to the `public` directory of the host application.

Simplify the module configuration

File `module.config.php` of `ZfDeals` already looks a bit messy. It holds all route definitions as well as the ones for services and controllers. To make it more readable, I move some definitions out of the file. First, I move all service definitions to `services.config.php`:

```php
<?php
return array(
        'factories' => array(
                        'ZfDeals\Mapper\Product' => function ($sm) {
                                return new \ZfDeals\Mapper\Product(
                                $sm->get('Zend\Db\Adapter\Adapter')
                        );
                },
                'ZfDeals\Mapper\Deal' => function ($sm) {
                        return new \ZfDeals\Mapper\Deal(
                                $sm->get('Zend\Db\Adapter\Adapter')
                        );
                },
                'ZfDeals\Mapper\Order' => function ($sm) {
                        return new \ZfDeals\Mapper\Order(
                                $sm->get('Zend\Db\Adapter\Adapter')
                        );
                },
                'ZfDeals\Validator\DealAvailable' => function ($sm) {
                        $validator = new \ZfDeals\Validator\DealActive();
                        $validator->setDealMapper($sm->get('ZfDeals\Mapper\Deal'));

                        $validator->setProductMapper(
                                $sm->get('ZfDeals\Mapper\Product')
                        );

                        return $validator;
                },
                'ZfDeals\Service\Checkout' => function ($sm) {
                        $srv = new \ZfDeals\Service\Checkout();

                        $srv->setDealAvailable(
                                $sm->get('ZfDeals\Validator\DealAvailable')
                        );

                        $srv->setProductMapper($sm->get('ZfDeals\Mapper\Product'));
                        $srv->setOrderMapper($sm->get('ZfDeals\Mapper\Order'));
                        $srv->setDealMapper($sm->get('ZfDeals\Mapper\Deal'));
                        return $srv;
                },
        ),
);
```

Listing 26.48

All controller definitions go to `controllers.config.php`:

```php
<?php
return array(
        'invokables' => array(
                'ZfDeals\Controller\Admin' => 'ZfDeals\Controller\AdminController',
        ),
        'factories' => array(
                'ZfDeals\Controller\CheckoutForm' => function ($serviceLocator) {
                        $form = new \ZfDeals\Form\Checkout();
                        $ctr = new ZfDeals\Controller\CheckoutFormController($form);

                        $productMapper = $serviceLocator
                                ->getServiceLocator()->get('ZfDeals\Mapper\Product');

                        $ctr->setProductMapper($productMapper);

                        $validator = $serviceLocator->getServiceLocator()
                                ->get('ZfDeals\Validator\DealAvailable');

                        $ctr->setdealActiveValidator($validator);

                        $checkoutService = $serviceLocator
                                ->getServiceLocator()->get('ZfDeals\Service\Checkout');

                        $ctr->setCheckoutService($checkoutService);
                        return $ctr;
                },
                'ZfDeals\Controller\DealAddForm' => function ($serviceLocator) {
                        $form = new ZfDeals\Form\DealAdd();
                        $ctr = new ZfDeals\Controller\DealAddFormController($form);

                        $dealMapper = $serviceLocator
                                ->getServiceLocator()->get('ZfDeals\Mapper\Deal');

                        $ctr->setDealMapper($dealMapper);

                        $productMapper = $serviceLocator
                                ->getServiceLocator()->get('ZfDeals\Mapper\Product');

                        $ctr->setProductMapper($productMapper);
                        return $ctr;
                },
                'ZfDeals\Controller\ProductAddForm' => function ($serviceLocator) {
                        $form = new \ZfDeals\Form\ProductAdd();
                        $ctr = new ZfDeals\Controller\ProductAddFormController($form);

                        $productMapper = $serviceLocator
                                ->getServiceLocator()->get('ZfDeals\Mapper\Product');

                        $ctr->setProductMapper($productMapper);
                        return $ctr;
                },
                'ZfDeals\Controller\Index' => function ($serviceLocator) {
                        $ctr = new ZfDeals\Controller\IndexController();

                        $productMapper = $serviceLocator
                                ->getServiceLocator()
                                ->get('ZfDeals\Mapper\Product');

                        $dealMapper = $serviceLocator->
                                getServiceLocator()->
                                get('ZfDeals\Mapper\Deal');

                        $ctr->setDealMapper($dealMapper);
                        $ctr->setProductMapper($productMapper);
                        return $ctr;
                },
                'ZfDeals\Controller\Order' => function ($serviceLocator) {
                        $ctr = new ZfDeals\Controller\OrderController();

                        $ctr->setOrderMapper($serviceLocator
                                ->getServiceLocator()->get('ZfDeals\Mapper\Order'));

                        return $ctr;
                },
        ),
);
```

Listing 26.49

I add the following methods to class `Module` to include the additional configuration files:

```php
<?php
// [..]
public function getServiceConfig()
{
        return include __DIR__ . '/config/services.config.php';
}

public function getControllerConfig()
{
        return include __DIR__ . '/config/controllers.config.php';
}
// [..]
```

Listing 26.50

File `module.config.php` itself now only holds the route definitions and view configuration.

A better approach for displaying forms

I can also tweak form rendering. Currently, in all three `show.phtml` templates, I mainly have the code present:

```php
<?php
$this->form->prepare();
echo $this->form()->openTag($this->form);
echo $this->formRowTwb($this->form->get('product')->get('id'));
echo $this->formRowTwb($this->form->get('product')->get('name'));
echo $this->formRowTwb($this->form->get('product')->get('stock'));
echo $this->formSubmitTwb($this->form->get('submit'));
echo $this->form()->closeTag();
```

Listing 26.51

Also, if I add another field to my form, I will also need change the template to make a new field appear. A custom "View Helper" can help here. The following lines of code render a form dynamically, based on the form definition. It makes form rendering a one-liner:

```php
<?php
echo $this->renderForm($form);
```

Listing 26.52

`RenderForm` itself looks like this:

```php
<?php
namespace ZfDeals\View\Helper;

use Zend\View\Helper\AbstractHelper;

class RenderForm extends AbstractHelper
{
        public function __invoke($form)
        {
                $form->prepare();
                $html = $this->view->form()->openTag($form) . PHP_EOL;
                $html .= $this->renderFieldsets($form->getFieldsets());
                $html .= $this->renderElements($form->getElements());
                $html .= $this->view->form()->closeTag($form) . PHP_EOL;
                return $html;
        }

        private function renderFieldsets($fieldsets)
        {
                $html = '';

                foreach($fieldsets as $fieldset)
                {
                        if(count($fieldset->getFieldsets()) > 0) {
                                $html .= $this->renderFieldsets(
                                        $fieldset->getFieldsets()
                                );
                        }

                        $html .= $this->renderElements(
                                $fieldset->getElements()
                        );
                }

                return $html;
        }

        private function renderElements($elements)
        {
                $html = '';

                foreach($elements as $element) {
                        $html .= $this->renderElement($element);
                }

                return $html;
        }

        private function renderElement($element)
        {
                if($element->getAttribute('type') == 'submit') {
                        return $this->view->formSubmitTwb($element) . PHP_EOL;
                } else {
                return $this->view->formRow($element) . PHP_EOL;
                }
        }
}
```

Listing 26.53

An important aspect of `RenderForm` is that it handles fieldsets by using recursion. `RenderForm` needs to be declared before it can be used in views. Again, to keep the module configuration slim, its declaration goes into a separate file called `viewhelper.config.php`:

```php
<?php
return array(
        'invokables' => array(
                'renderForm' => 'ZfDeals\View\Helper\RenderForm'
        )
);
```

Listing 26.54

Method `getViewHelperConfig()` in class `Module` makes it available to the module:

```php
<?php
// [..]
public function getViewHelperConfig()
{
    return include __DIR__ . '/config/viewhelper.config.php';
}
// [..]
```

Listing 26.55

Now ZfDeals is ready to be used in 3rd party applications!

What's next?

Granted, ZfDeals is not yet feature rich, but already fully functional and somewhat helpful. We can now go from there and add feature by feature. In addition, there is some more refactoring I could do one day. Let's take a look!

Zend\Di for Dependency Injection

I wrote a lot of code to apply Dependency Injection to my application. Most of the hand coded factories look way more complicated than they should. Mainly, they all simply create a service or controller by only retrieving other services and injecting them using "setter methods". All this feels like I violate the DRY principal[92]. Do I really need to repeat myself and write dedicated factories to only inject dependencies? I may think about using `Zend\Di`. Doing so could more or less eliminate all of my hand coded factories.

Doctrine 2 for data persistence

Next, a huge part of my code deals with reading data from and writing data to the database. A system like Doctrine 2 ORM can nearly eliminate all this custom code making data persistence fully transparent to the application developer. Just deal with your PHP objects and let Doctrine 2 take care of all the persistence work.

Utilize ZF2's event system

Currently, I assume that ZfDeals is used "as is". It nearly has no options for customization. But what if, for instance, one wants to send an order notification to customer service? To make ZfDeals extendable, I may introduce ZF2's event system and trigger proper events when they occur. This would allow developers to attach event based custom code to extend or modify ZfDeals's processing logic.

Improve static assets handling

Copying over static assets from the module to the host applications `public` folder is error-prone and inconvenient. I may introduce a proper "Asset Manager" like zf2-module-assets[93] or "Assetic" with its ZF2 "glue code module" zf2-assetic-module[94] allowing to serve static assets from within the modules directly.

[92]http://de.wikipedia.org/wiki/Don%E2%80%99t_repeat_yourself
[93]https://github.com/albulescu/zf2-module-assets
[94]https://github.com/widmogrod/zf2-assetic-module

Source Code download

You can find the Developer's Diary source code, the host application as well as module code, on GitHub in Repository "ZfDealsApp"[a] and Repository "ZfDeals"[b].

[a] https://github.com/michael-romer/ZfDealsApp

[b] https://github.com/michael-romer/ZfDeals