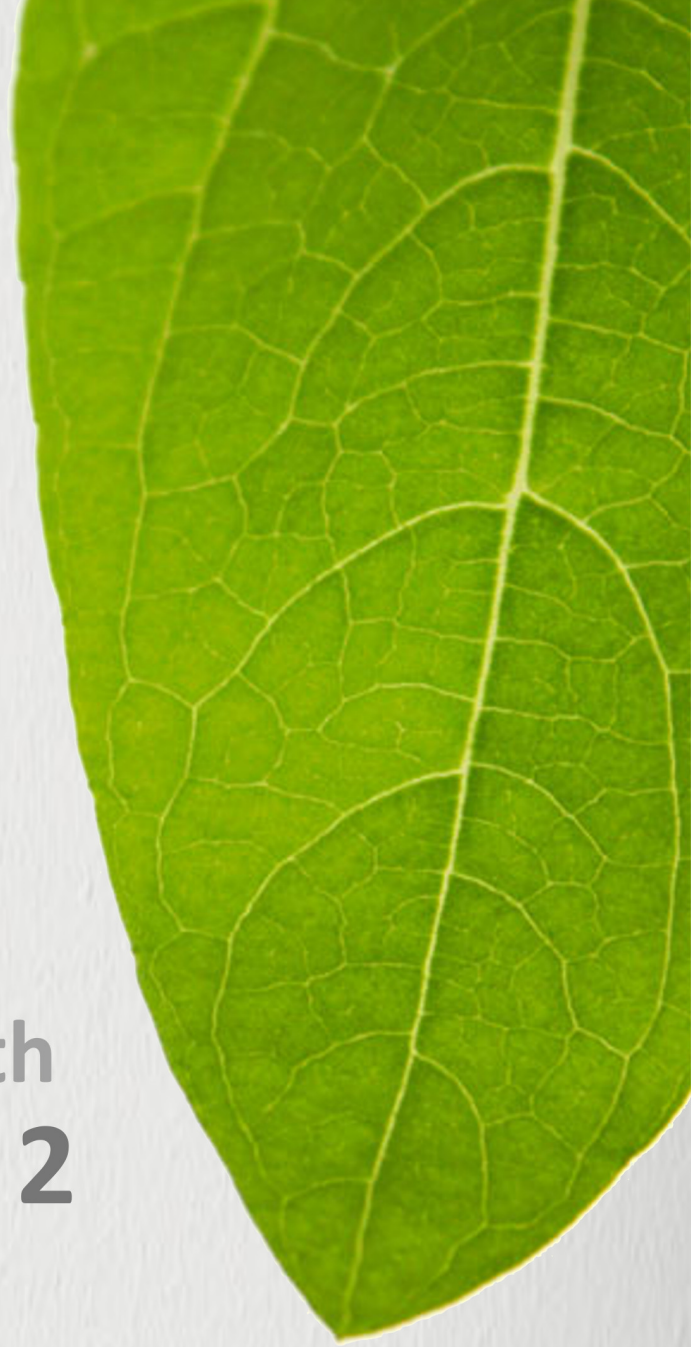


Zend Framework
Version 2.2

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

This book is for sale at http://leanpub.com/zendframework2-en

This version was published on 2013-08-15

Also By Michael Romer

Webentwicklung mit Zend Framework 2

Persistenz in PHP mit Doctrine 2 ORM

PHP Data Persistence with Doctrine 2 ORM

Developing Apps for Firefox OS

Contents

1 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.

1.1 Installation

In principle, Zend Framework 2 does not have to be tediously installed. One simply downloads the Code[1], 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.

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

1.2 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[2] is available for downloading. The installation under Linux nearly always runs under the respective package manager. After installation and after invocation of

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

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

```
1 > 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 .

```
1 $ 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.

1.3 Composer

The "ZendSkeletonApplication" uses with Composer[3] 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.

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

[3] 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[4]

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.

```
1 $ cd ZendSkeletonApplication
2 $ php composer.phar install
3 > 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[5], 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.

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

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

Phar Archive and Suhosin

If "Suhosin""[6] 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.

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[7].

1.4 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:

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

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

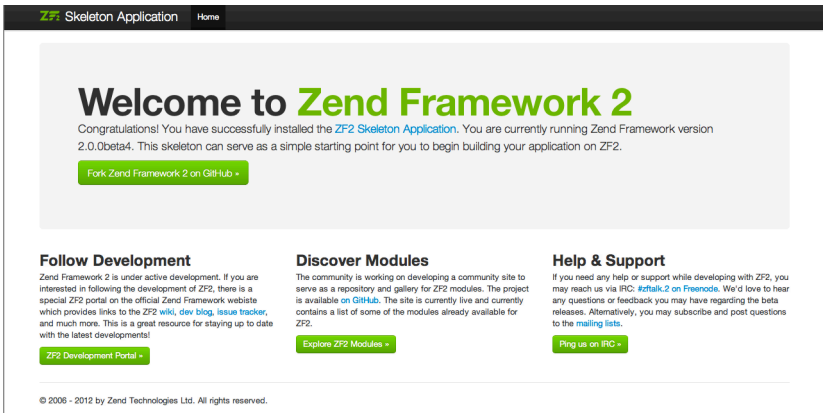
```
1 $ cd /var/www
2 $ git clone git://github.com/zendframework/ZendSkeletonApplication.git
```

[6] http://www.hardened-php.net/suhosin/
[7] http://framework.zend.com/downloads

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:



Start page ZendSkeletonApplication of Zend Framework 2

1.5 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:

```
1  ZendSkeletonApplication/
2          config/
3              application.config.php
4              autoload/
5                  global.php
6                  local.php
7                  ...
8          module/
9          vendor/
```

```
10          public/
11              .htaccess
12              index.php
13      data/
```

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:

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

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[8], 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`'.

[8] 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.

1.6 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
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
// [..]
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:

```
1 <?php
2 // [..]
3 Zend\Mvc\Application::init(
4         include 'config/application.config.php')
5         ->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
// [..]
$serviceManager = new ServiceManager(
        new ServiceManagerConfig($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
// [..]
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
// [..]
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[9]—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”.

[9] http://nginx.org/