RogueWave SOFTWARE

zend®

ZEND FRAMEWORK 3 COOKBOOK

A collection of PHP recipes using Zend components

Table of Contents

Zend Framework 3 Cookbook

During the year 2017, Matthew Weier O'Phinney and Enrico Zimuel started a series of blog posts on the offical Zend Framework blog covering its components.

Zend Framework is composed by 60+ components covering a wide range of functionality. While the framework has typically been marketed as a full-stack MVC framework, the individual components themselves typically work independently and can be used standalone or within other frameworks. The blog posts were written to highlight this fact, and demonstrate how to get started with a number of the more popular and useful components.

We hope this book will help you get started using Zend Framework components, no matter what project you are writing!

Enjoy your reading,
Matthew Weier O'Phinney and *Enrico Zimuel*
Rogue Wave Software, Inc.

About the authors

Matthew Weier O'Phinney is a *Principal Engineer* at Rogue Wave Software, and project lead for the *Zend Framework*, *Apigility*, and *Expressive* projects. He's responsible for architecture, planning, and community engagement for each project, which are used by thousands of developers worldwide, and shipped in projects from personal websites to multinational media conglomerates, and everything in between. When not in front of a computer, you'll find him with his family and dogs on the plains of South Dakota.

For more information:

- https://mwop.net/
- https://www.roguewave.com/

Enrico Zimuel has been a *software developer* since 1996. He works as a *Senior Software Engineer* at Rogue Wave Software as a core developer of the *Zend Framework*, *Apigility*, and *Expressive* projects. He is a former *Researcher Programmer* for the Informatics Institute of the University of Amsterdam. Enrico speaks regularly at conferences and events, including TEDx and international PHP conferences. He is also the co-founder of the PHP User Group of Torino (Italy).

For more information:

- https://www.zimuel.it/

- https://www.roguewave.com/
- TEDx presentation: https://www.youtube.com/watch?v=SienrLY40-w
- PHP User Group of Torino: http://torino.grusp.org/

Zend-config for all your configuration needs

by Matthew Weier O'Phinney

Different applications and frameworks have different opinions about how configuration should be created. Some prefer XML, others YAML, some like JSON, others like INI, and some even stick to the JavaProperties format; in Zend Framework, we tend to prefer PHP arrays, as each of the other formats essentially get compiled to PHP arrays eventually anyways.

At heart, though, we like to support developer needs, whatever they may be, and, as such, our zend-config component[1] provides ways of working with a variety of configuration formats.

Installation

zend-config is installable via Composer:

```
$ composer require zendframework/zend-config
```

The component has two dependencies:

- zend-stdlib[2], which provides some capabilities around configuration merging.
- psr/container[3], to allow reader and writer plugin support for the configuration factory.

> ### Latest version
>
> This article covers the most recently released version of zend-config, 3.1.0, which contains a number of features such as PSR-11 support that were not previously available. If you are using Zend Framework MVC layer, you should be able to safely provide the constraint `^2.6 || ^3.1`, as the primary APIs remain the same.

Retrieving configuration

Once you've installed zend-config, you can start using it to retrieve and access configuration files. The simplest way is to use `Zend\Config\Factory`, which provides tools for loading configuration from a variety of formats, as well as capabilities for merging.

If you're just pulling in a single file, use `Factory::fromFile()` :

```
use Zend\Config\Factory;

$config = Factory::fromFile($path);
```

Far more interesting is to use multiple files, which you can do via `Factory::fromFiles()` . When you do, they are merged into a single configuration, in the order in which they are provided to the factory. This is particularly interesting using `glob()` :

```
use Zend\Config\Factory;

$config = Factory::fromFiles(glob('config/autoload/*.*'));
```

This method supports a variety of formats:

- PHP files returning arrays (`.php` extension)
- INI files (`.ini` extension)
- JSON files (`.json` extension)
- XML files (using PHP's `XMLReader` ; `.xml` extension)
- YAML files (using ext/yaml, installable via PECL; `.yaml` extension)
- JavaProperties files (`.javaproperties` extension)

This means that you can choose the configuration format you prefer, or mix-and-match multiple formats, if you need to combine configuration from multiple libraries!

Configuration objects

By default, `Zend\Config\Factory` will return PHP arrays for the merged configuration. Some dependency injection containers do not support arrays as services, however; moreover, you may want to pass some sort of structured object instead of a plain array when injecting dependencies.

As such, you can pass a second, optional argument to each of `fromFile()` and `fromFiles()` , a boolean flag. When `true` , it will return a `Zend\Config\Config` instance, which implements `Countable` , `Iterator` , and `ArrayAccess` , allowing it to look and act like an array.

What is the benefit?

First, it provides property overloading to each configuration key:

```php
$debug = $config->debug ?? false;
```

Second, it offers a convenience method, `get()`, which allows you to specify a default value to return if the value is not found:

```php
$debug = $config->get('debug', false); // Return false if not found
```

This is largely obviated by the `??` ternary shortcut in modern PHP versions, but very useful when *mocking* in your tests.

Third, nested sets are also returned as `Config` instances, which gives you the ability to use the above `get()` method on a nested item:

```php
if (isset($config->expressive)) {
    $config = $config->get('expressive'); // same API!
}
```

Fourth, you can mark the `Config` instance as immutable! By default, it acts just like array configuration, which is, of course, mutable. However, this can be problematic when you use configuration as a service, because, unlike an array, a `Config` instance is passed by reference, and changes to values would then propagate to any other services that depend on the configuration.

Ideally, you wouldn't be changing any values in the instance, but `Zend\Config\Config` can enforce that for you:

```php
$config->setReadOnly(); // Now immutable!
```

Further, calling this will mark nested `Config` instances as read-only as well, ensuring data integrity for the entire configuration tree.

> ### Read-only by default!
>
> One thing to note: by default, `Config` instances are read-only! The constructor accepts an optional, second argument, a flag indicating whether or not the instance allows modifications, and the value is `false` by default. When you use the `Factory` to create a `Config` instance, it never enables that flag, meaning that if you return a `Config` instance, it will be read-only.
>
> If you want a mutable instance from a `Factory`, use the following construct:
>
> ```
> use Zend\Config\Config;
> use Zend\Config\Factory;
>
> $config = new Config(Factory::fromFiles($files), true);
> ```

Including other configuration

Most of the configuration reader plugins also support "includes": directives within a configuration file that will include configuration from another file. (JavaProperties is the only configuration format we support that does not have this functionality included.)

For instance:

- INI files can use the key `@include` to include another file relative to the current one; values are merged at the same level:

  ```
  webhost = 'www.example.com'
  @include = 'database.ini'
  ```

- For XML files, you can use XInclude:

  ```
  <?xml version="1.0" encoding="utf-8">
  <config xmlns:xi="http://www.w3.org/2001/XInclude">
    <webhost>www.example.com</webhost>
    <xi:include href="database.xml"/>
  </config>
  ```

- JSON files can use an `@include` key:

  ```
  {
    "webhost": "www.example.com",
    "@include": "database.json"
  }
  ```

- YAML also uses the `@include` notation:

  ```
  webhost: www.example.com
  @include: database.yaml
  ```

Choose your own YAML

Out-of-the-box we support the YAML PECL extension for our YAML support. However, we have made it possible to use alternate parsers, such as Spyc or the Symfony YAML component, by passing a callback to the reader's constructor:

```
use Symfony\Component\Yaml\Yaml as SymfonyYaml;
use Zend\Config\Reader\Yaml as YamlConfig;

$reader = new YamlConfig([SymfonfyYaml::class, 'parse']);
$config = $reader->fromFile('config.yaml');
```

Of course, if you're going to do that, you could just use the original library, right? But what if you want to mix YAML and other configuration with the `Factory` class?

There are two ways to register new plugins. One is to create an instance and register it with the factory:

```
use Symfony\Component\Yaml\Yaml as SymfonyYaml;
use Zend\Config\Factory;
use Zend\Config\Reader\Yaml as YamlConfig;

Factory::registerReader('yaml', new YamlConfig([SymfonyYaml::class, 'parse']));
```

Alternately, you can provide an alternate reader plugin manager. You can do that by extending `Zend\Config\StandaloneReaderPluginManager`, which is a barebones PSR-11 container for use as a plugin manager:

```
namespace Acme;

use Symfony\Component\Yaml\Yaml as SymfonyYaml;
use Zend\Config\Reader\Yaml as YamlConfig;
use Zend\Config\StandaloneReaderPluginManager;

class ReaderPluginManager extends StandaloneReaderPluginManager
{
    /**
     * @inheritDoc
     */
    public function has($plugin)
    {
        if (YamlConfig::class === $plugin
            || 'yaml' === strtolower($plugin)
        ) {
            return true;
        }

        return parent::has($plugin);
    }

    /**
     * @inheritDoc
     */
    public function get($plugin)
    {
        if (YamlConfig::class !== $plugin
            && 'yaml' !== strtolower($plugin)
        ) {
            return parent::get($plugin);
        }

        return new YamlConfig([SymfonyYaml::class, 'parse']);
    }
}
```

Then register this with the `Factory`:

```
use Acme\ReaderPluginManager;
use Zend\Config\Factory;

Factory::setReaderPluginManager(new ReaderPluginManager());
```

Processing configuration

zend-config also allows you to *process* a `Zend\Config\Config` instance and/or an individual value. Processors perform operations such as:

- substituting constant values within strings
- filtering configuration data
- replacing tokens within configuration
- translating configuration values

Why would you want to do any of these operations?

Consider this: deserialization of formats other than PHP cannot take into account PHP constant values or class names!

While this may work in PHP:

```php
return [
    Acme\Component::CONFIG_KEY => [
        'host' => Acme\Component::CONFIG_HOST,
        'dependencies' => [
            'factories' => [
                Acme\Middleware\Authorization::class => Acme\Middleware\AuthorizationF
actory::class,
            ],
        ],
    ],
];
```

The following JSON configuration would not:

```json
{
    "Acme\\Component::CONFIG_KEY": {
        "host": "Acme\\Component::CONFIG_HOST"
        "dependencies": {
            "factories": {
                "Acme\\Middleware\\Authorization::class": "Acme\\Middleware\\Authoriza
tionFactory::class"
            }
        }
    }
}
```

Enter the `Constant` processor!

This processor looks for strings that match constant names, and replaces them with their values. Processors generally only work on the configuration *values*, but the `Constant` processor allows you to opt-in to processing the *keys* as well.

Since processing *modifies* the `Config` instance, you will need to manually create an instance, and then process it. Let's look at that:

```
use Acme\Component;
use Zend\Config\Config;
use Zend\Config\Factory;
use Zend\Config\Processor;

$config = new Config(Factory::fromFile('config.json'), true);
$processor = new Processor\Constant();
$processor->enableKeyProcessing();
$processor->process($config);
$config->setReadOnly();

var_export($config->{Component::CONFIG_KEY}->dependencies->factories);
// ['Acme\Middleware\Authorization' => 'Acme\Middleware\AuthorizationFactory']
```

This is a really powerful feature, as it allows you to add more verifications and validations to your configuration files, regardless of the format you use.

> ### In version 3.1.0 forward
>
> The ability to work with class constants and process keys was added starting with the 3.1.0 version of zend-config.

Config all the things!

This post covers the parsing features of zend-config, but does not even touch on another major capability: the ability to *write* configuration! We'll leave that to another post.

In terms of configuration parsing, zend-config is simple, yet powerful. The ability to process a number of common configuration formats, utilize configuration includes, and process keys and values means you can highly customize your configuration process to suit your needs or integrate different configuration sources.

Get more information from the zend-config documentation[4].

Footnotes

[1]. https://docs.zendframework.com/zend-config/ ↩

[2]. https://docs.zendframework.com/zend-stdlib/ ↩

[3]. https://github.com/php-fig/container ↩

[4]. https://docs.zendframework.com/zend-config/ ↩

Manage your application with zend-config-aggregator

by Matthew Weier O'Phinney

With the rise of PHP middleware, many developers are creating custom application architectures, and running into an issue many frameworks already solve: how to allow runtime configuration of the application.

configuration is often necessary, even in custom applications:

- Some configuration, such as API keys, may vary between environments.
- You may want to substitute services between development and production.
- Some code may be developed by other teams, and pulled into your application separately (perhaps via Composer[1]), and require configuration.
- You may be writing code in your application that you will later want to share with another team, and recognize it should provide service wiring information or allow for dynamic configuration itself.

Faced with this reality, you then have a new problem: how can you configure your application, as well as aggregate configuration from other sources?

As part of the Expressive initiative, we now offer a standalone solution for you: zend-config-aggregator[2].

Installation

First, you will need to install zend-config-aggregator:

```
$ composer require zendframework/zend-config-aggregator
```

One feature of zend-config-aggregator is the ability to consume multiple configuration formats via zend-config[3]. If you wish to use that feature, you will also need to install that package:

```
$ composer require zendframework/zend-config
```

Finally, if you are using the above, and want to parse YAML files, you will need to install the YAML PECL extension[4].

Configuration providers

zend-config-aggregator allows you to aggregate configuration from configuration *providers*. A configuration provider is any PHP callable that will return an associative array of configuration.

By default, the component provides the following providers out of the box:

- `Zend\ConfigAggregator\ArrayProvider`, which accepts an array of configuration and simply returns it. This is primarily useful for providing global defaults for your application.
- `Zend\ConfigAggregator\PhpFileProvider`, which accepts a glob pattern describing PHP files that each return an associative array. When invoked, it will loop through each file, and merge the results with what it has previously stored.
- `Zend\ConfigAggregator\ZendConfigProvider`, which acts similarly to the `PhpFileProvider`, but which can aggregate any format zend-config supports, including INI, XML, JSON, and YAML.

More interestingly, however, is the fact that you can write providers as simple invokable objects:

```
namespace Acme;

class ConfigProvider
{
    public function __invoke()
    {
        return [
            // associative array of configuration
        ];
    }
}
```

This feature allows you to write configuration for specific application *features*, and then seed your application with it. In other words, this feature can be used as the foundation for a *modular architecture*[5], which is exactly what we did with Expressive!

> ### Generators
>
> You may also use invokable classes or PHP callables that define generators as configuration providers! As an example, the `PhpFileProvider` could potentially be rewritten as follows:
>
> ```
> use Zend\Stdlib\Glob;
>
> function () {
>     foreach (Glob::glob('config/*.php', Glob::GLOB_BRACE) as $file) {
>         yield include $file;
>     }
> }
> ```

Aggregating configuration

Now that you have configuration providers, you can aggregate them.

For the purposes of this example, we'll assume the following:

- We will have a single configuration file, `config.php`, at the root of our application which will aggregate all other configuration.
- We have a number of configuration files under `config/`, including YAML, JSON, and PHP files.
- We have a third-party "module" that exposes the class `Umbrella\ConfigProvider`.
- We have developed our own "module" for re-distribution that exposes the class `Blanket\ConfigProvider`.

Typically, you will want aggregate configuration such that third-party configuration is loaded first, with application-specific configuration merged last, in order to override settings.

Let's aggregate and return our configuration.

```
// in config.php:
use Zend\ConfigAggregator\ConfigAggregator;
use Zend\ConfigAggregator\ZendConfigProvider;

$aggregator = new ConfigAggregator([
    \Umbrella\ConfigProvider::class,
    \Blanket\ConfigProvider::class,
    new ZendConfigProvider('config/*.{json,yaml,php}'),
]);

return $aggregator->getMergedConfig();
```

This file aggregates the third-party configuration provider, the one we expose in our own application, and then aggregates a variety of different configuration files in order to, in the end, return an associative array representing the merged configuration!

> ### Valid config profider entries
>
> You'll note that the `ConfigAggregator` expects an array of providers as the first argument to the constructor. This array may consist of any of the following:
>
> - Any PHP callable (functions, invokable objects, closures, etc.) returning an array.
> - A class name of a class that defines `__invoke()`, and which requires no constructor arguments.
>
> This latter is useful, as it helps reduce operational overhead once you introduce caching, which we discuss below. The above example demonstrates this usage.
>
> ### zend-config and PHP configuration
>
> The above example uses only the `ZendConfigProvider`, and not the `PhpFileProvider`. This is due to the fact that zend-config can also consume PHP configuration.
>
> If you are only using PHP-based configuration files, you can use the `PhpFileProvider` instead, as it does not require additionally installing the zendframework/zend-config package.
>
> ### Globbing and precedence
>
> Globbing works as it does on most *nix systems. As such, you need to pay particular attention to when you use patterns that define alternatives, such as the `{json,yaml,php}` pattern above. In such cases, all JSON files will be aggregated, followed by YAML files, and finally PHP files. If you need them to aggregate in a different order, you will need to change the pattern.

Caching

You likely do not want to aggregate configuration on each and every application request, particularly if doing so would result in many filesystem hits. Fortunately, zend-config-aggregator also has built-in caching features.

To enable these features, you will need to do two things:

- First, you need to provide a second argument to the `ConfigAggregator` constructor,

specifying the path to the cache file to create and/or use.
- Second, you need to enable caching in your configuration, by specifying a boolean `true` value for the key `ConfigAggregator::ENABLE_CACHE`.

One common strategy is to enable caching by default, and then disable it via environment-specific configuration.

We'll update the above example now to enable caching to the file `cache/config.php`:

```php
use Zend\ConfigAggregator\ArrayProvider;
use Zend\ConfigAggregator\ConfigAggregator;
use Zend\ConfigAggregator\PhpFileProvider;
use Zend\ConfigAggregator\ZendConfigProvider;

$aggregator = new ConfigAggregator(
    [
        new ArrayProvider([ConfigAggregator::ENABLE_CACHE => true]),
        \Umbrella\ConfigProvider::class,
        \Blanket\ConfigProvider::class,
        new ZendConfigProvider('config/{,*.}global.{json,yaml,php}'),
        new PhpFileProvider('config/{,*.}local.php'),
    ],
    'cache/config.php'
);

return $aggregator->getMergedConfig();
```

The above adds an initial setting that enables the cache, and tells it to cache it to `cache/config.php`.

Notice also that this example changes the `ZendConfigProvider`, and adds a `PhpFileProvider` entry. Let's examine these.

The `ZendConfigProvider` glob pattern now looks for files named `global` with one of the accepted extensions, or those named `*.global` with one of the accepted extensions. This allows us to segregate configuration that should *always* be present from environment-specific configuration.

We then add a `PhpFileProvider` that aggregates `local.php` and/or `*.local.php` files specifically. An interesting side-note about the shipped providers is that if no matching files are found, the provider will return an empty array; this means that we can have this additional provider that is looking for separate configurations for the "local" environment! Because this provider is aggregated last, the settings it exposes will override any others.

As such, if we want to *disable* caching, we can create a file such as `config/local.php` with the following contents:

```php
<?php
use Zend\ConfigAggregator\ConfigAggregator;

return [ConfigAggregator::ENABLE_CACHE => false];
```

and the application will no longer cache aggregated configuration!

> ### Clear the cache!
>
> The setting outlined above is used to determine whether the configuration cache file *should be created if it does not already exist*. zend-config-aggregator, when provided the location of a configuration cache file, will load directly from it if the file is present.
>
> As such, if you make the above configuration change, you will first need to remove any cached configuration:
>
> ```
> $ rm cache/config.php
> ```
>
> This can even be made into a Composer script:
>
> ```json
> "scripts": {
>     "clear-config-cache": "rm cache/config.php"
> }
> ```
>
> Allowing you to do this:
>
> ```
> $ composer clear-config-cache
> ```
>
> Which allows you to change the location of the cache file without needing to re-learn the location every time you need to clear the cache.

Auto-enabling third-party providers

Being able to aggregate providers from third-parties is pretty stellar; it means that you can be assured that configuration the third-party code expects is generally present — with the exception of values that *must* be provided by the consumer, that is!

However, there's one minor problem: you need to remember to register these configuration providers with your application, by manually editing your `config.php` file and adding the appropriate entries.

[6]

Zend Framework solves this via the zf-component-installer Composer plugin[6]. If your package is installable via Composer, you can add an entry to your package definition as follows:

```
"extra": {
    "zf": {
        "config-provider": [
            "Umbrella\\ConfigProvider"
        ]
    }
}
```

If the end-user:

- Has required `zendframework/zend-component-installer` in their application (as either a production or development dependency), **AND**
- has the config aggregation script in `config/config.php`

then the plugin will prompt you, asking if you would like to add each of the `config-provider` entries found in the installed package into the configuration script.

As such, for our example to work, we would need to move our configuration script to `config/config.php`, and likely move our other configuration files into a sub-directory:

```
cache/
    config.php
config/
    config.php
    autoload/
        blanket.global.yaml
        global.php
        umbrella.global.json
```

This approach is essentially that taken by Expressive.

When those changes are made, any package you add to your application that exposes configuration providers will prompt you to add them to your configuration aggregation, and, if you confirm, will add them to the top of the script!

Final notes

First, we would like to thank Mateusz Tymek[7], whose prototype 'expressive-config-manager' project became zend-config-aggregator. This is a stellar example of a community project getting adopted into the framework!

Second, this approach has some affinity to a proposal from the folks who brought us PSR-11, which defines the `ContainerInterface` used within Expressive for allowing usage of different dependency injection containers. That same group is now working on a service provider[8] proposal that would standardize how standalone libraries expose services to containers; we recommend looking at that project as well.

We hope that this post helps spawn ideas for configuring your next project!

Footnotes

> [1]. https://getcomposer.org ↩
>
> [2]. https://github.com/zendframework/zend-config-aggregator ↩
>
> [3]. https://docs.zendframework.com/zend-config/ ↩
>
> [4]. http://www.php.net/manual/en/book.yaml.php ↩
>
> [5]. https://docs.zendframework.com/zend-expressive/features/modular-applications/ ↩
>
> [6]. https://docs.zendframework.com/zend-component-installer/ ↩
>
> [7]. http://mateusztymek.pl/ ↩
>
> [8]. https://github.com/container-interop/service-provider ↩

Convert objects to arrays and back with zend-hydrator

by Matthew Weier O'Phinney

APIs are all the rage these days, and a tremendous number of them are being written in PHP. When APIs were first gaining popularity, this seemed like a match made in heaven: query the database, pass the results to `json_encode()`, and voilà! API payload! In reverse, it's `json_decode()`, pass the data to the database, and done!

Modern day professional PHP, however, is skewing towards usage of value objects and entities, but we're still creating APIs. How can we take these objects and create our API response payloads? How can we take incoming data and transform it into the domain objects we need?

Zend Framework's answer to that question is zend-hydrator. Hydrators can *extract* an associative array of data from an object, and *hydrate* an object from an associative array of data.

Installation

As with our other components, you can install zend-hydrator by itself:

```
$ composer require zendframework/zend-hydrator
```

Out-of-the-box, it only requires zend-stdlib, which is used internally for transforming iterators to associative arrays. However, there are a number of other interesting, if optional, features that require other components:

- You can create an *aggregate* hydrator where each hydrator is responsible for a subset of data. This requires zend-eventmanager.
- You can filter/normalize the keys/properties of data using *naming strategies*; these require zend-filter.
- You can map object types to hydrators, and delegate hydration of arbitrary objects using the `DelegatingHydrator`. This feature utilizes the provided `HydratorPluginManager`, which requires zend-servicemanager.

In our examples below, we'll be demonstrating naming strategies and the delegating hydrator, so we will install the dependencies those need:

```
$ composer require zendframework/zend-filter zendframework/zend-servicemanager
```

Objects to arrays and back again

Let's take the following class definition:

```
namespace Acme;

class Book
{
    private $id;

    private $title;

    private $author;

    public function __construct(int $id, string $title, string $author)
    {
        $this->id = $id;
        $this->title = $title;
        $this->author = $author;
    }
}
```

What we have is a value object, with no way to publicly grab any given datum. We now want to represent it in our API. How do we do that?

The answer is via reflection, and zend-hydrator provides a solution for that:

```
use Acme\Book;
use Zend\Hydrator\Reflection as ReflectionHydrator;

$book = new Book(42, 'Hitchhiker\'s Guide to the Galaxy', 'Douglas Adams');

$hydrator = new ReflectionHydrator();
$data = $hydrator->extract($book);
```

We now have an array representation of our `Book` instance!

Let's say that somebody has just submitted a book via a web form or an API. We have the values, but want to create a `Book` out of them.

```php
use Acme\Book;
use ReflectionClass;
use Zend\Hydrator\Reflection as ReflectionHydrator;

$hydrator = new ReflectionHydrator();
$book = $hydrator->hydrate(
    $incomingData,
    (new ReflectionClass(Book::class))->newInstanceWithoutConstructor()
);
```

And now we have a `Book` instance!

> The `newInstanceWithoutConstructor()` construct is necessary in this case because our class has required constructor arguments. Another possibility is to provide an already populated instance, and hope that the submitted data will overwrite all data in the class. Alternately, you can create classes that have optional constructor arguments.

Most of the time, it can be as simple as this: create an appropriate hydrator instance, and use either `extract()` to get an array representation of the object, or `hydrate()` to create an instance from an array of data.

We provide a number of standard implementations:

- `Zend\Hydrator\ArraySerializable` works with `ArrayObject` implementations. It will also hydrate any object implementing either the method `exchangeArray()` or `populate()`, and extract from any object implementing `getArrayCopy()`.
- `Zend\Hydrator\ClassMethods` will use setter and getter methods to populate and extract objects. It also understands `has*()` and `is*()` methods as getters.
- `Zend\Hydrator\ObjectProperty` will use public instance properties.
- `Zend\Hydrator\Reflection` can extract and populate instance properties of any visibility.

Filtering values

Since a common rationale for extracting data from objects is to create payloads for APIs, you may find there is data in your object you *do not* want to represent.

zend-hydrator provides a `Zend\Hydrator\Filter\FilterInterface` for accomplishing this. Filters implement the following:

```
namespace Zend\Hydrator\Filter;

interface FilterInterface
{
    /**
     * @param string $property
     * @return bool
    public function filter($property);
}
```

If a filter returns a boolean `true`, the value is kept; otherwise, it is omitted.

A `FilterComposite` implementation allows attaching multiple filters; each property is then checked against each filter. (This class also allows attaching standard PHP callables for filters, instead of `FilterInterface` implementations.) A `FilterEnabledInterface` allows a hydrator to indicate it composes filters. Tying it together, all shipped hydrators inherit from a common base that implements `FilterEnabledInterface` by composing a `FilterComposite`, which means that you can use filters immediately in a standard fashion.

As an example, let's say we have a `User` class that has a `password` property; we clearly do not want to return the password in our payload, even if it is properly hashed! Filters to the rescue!

```
use Zend\Hydrator\ObjectProperty as ObjectPropertyHydrator;

$hydrator = new ObjectPropertyHydrator();
$hydrator->addFilter('password', function ($property) {
    return $property !== 'password';
});
$data = $hydrator->extract($user);
```

Some hydrators actually use filters internally in order to do their work. As an example, the `ClassMethods` hydrator composes the following by default:

- `IsFilter`, to identify methods beginning with `is`, such as `isTransaction()`.
- `HasFilter`, to identify methods beginning with `has`, such as `hasAuthor()`.
- `GetFilter`, to identify methods beginning with `get`, such as `getTitle()`.
- `OptionalParametersFilter`, to ensure any given matched method can be executed without requiring any arguments.

This latter point brings up an interesting feature: since hydration runs each potential property name through each filter, you may need to setup rules. For example, with the `ClassMethods` hydrator, a given method name is valid if the following condition is met:

```
(matches "is" || matches "has" || matches "get") && matches "optional parameters"
```

As such, when calling `addFilter()`, you can specify an optional third argument: a flag indicating whether to `OR` or `AND` the given filter (using the values `FilterComposite::CONDITION_OR` or `FilterComposite::FILTER_AND`); the default is to `OR` the new filter.

Filtering is very powerful and flexible. If you remember only two things about filters:

- They only operate *during extraction*.
- They can only be used to determine what values to keep in the extracted data set.

Strategies

What if you wanted to alter the values returned during extraction or hydration? zend-hydrator provides these features via *strategies*.

A *strategy* provides functionality both for extracting and hydrating a value, and simply transforms it; think of strategies as normalization filters. Each implements `Zend\Hydrator\Strategy\StrategyInterface`:

```
namespace Zend\Hydrator\Strategy;

interface StrategyInterface
{
    public function extract($value;)
    public function hydrate($value;)
}
```

Like filters, a `StrategyEnabledInterface` allows a hydrator to indicate it accepts strategies, and the `AbstractHydrator` implements this interface, allowing you to use strategies out of the box with the shipped hydrators.

Using our previous `User` example, we could, instead of omitting the `password` value, instead return a static `********` value; a strategy could allow us to do that. Data submitted would be instead hashed using `password_hash()`:

```php
namespace Acme;

use Zend\Hydrator\Strategy\StrategyInterface;

class PasswordStrategy implements StrategyInterface
{
    public function extract($value)
    {
        return '********';
    }

    public function hydrate($value)
    {
        return password_hash($value);
    }
}
```

We would then extract our data as follows:

```php
use Acme\PasswordStrategy;
use Zend\Hydrator\ObjectProperty as ObjectPropertyHydrator;

$hydrator = new ObjectPropertyHydrator();
$hydrator->addStrategy('password', new PasswordStrategy());
$data = $hydrator->extract($user);
```

zend-hydrator ships with a number of really useful strategies for common data:

- `BooleanStrategy` will convert booleans into other values (such as `0` and `1`, or the strings `true` and `false`) and vice versa, according to a map you provide to the constructor.
- `ClosureStrategy` allows you to provide callbacks for each of extraction and hydration, allowing you to forego the need to create a custom strategy implementation.
- `DateTimeFormatterStrategy` will convert between strings and `DateTime` instances.
- `ExplodeStrategy` is a wrapper around `implode` and `explode()`, and expects a delimiter to its constructor.
- `StrategyChain` allows you to compose multiple strategies; the return value of each is passed as the value to the next, providing a filter chain.

Filtering property names

We can now filter properties to omit from our representations, as well as filter or normalize the values we ultimately want to represent. What about the property names, though?

In PHP, we often use `camelCase` to represent properties, but `snake_case` is typically more accepted for APIs. Additionally, what about when we use getters for our values? We likely don't want to use the actual method name as the property name!

For this reason, zend-hydrator provides *naming strategies*. These work just like strategies, but instead of working on the value, they work on the property name. Like both filters and strategies, an interface, `NamingStrategyEnabledInterface`, allows a hydrator to indicate can accept a naming strategy, and the `AbstractHydrator` implements that interface, to allow out of the box usage of naming strategies on the shipped hydrators.

As an example, let's consider the following class:

```php
namespace Acme;

class Transaction
{
    public $isPublished;
    public $publishedOn;
    public $updatedOn;
}
```

Let's now extract an instance of that class:

```php
use Acme\Transaction;
use Zend\Hydrator\NamingStrategy\UnderscoreNamingStrategy;
use Zend\Hydrator\ObjectProperty as ObjectPropertyHydrator;

$hydrator = new ObjectPropertyHydrator();
$hydrator->setNamingStrategy(new UnderscoreNamingStrategy());
$data = $hydrator->extract($transaction);
```

The extracted data will now have the keys `is_published`, `published_on`, and `updated_on`!

This is useful if you know all your properties will be camelCased, but what if you have other needs? For instance, what if you want to rename `isPublished` to `published` instead?

A `CompositeNamingStrategy` class allows you to do exactly that. It accepts an associative array of object property names mapped to the naming strategy to use with it. So, as an example:

```
use Acme\Transaction;
use Zend\Hydrator\NamingStrategy\CompositeNamingStrategy;
use Zend\Hydrator\NamingStrategy\MapNamingStrategy;
use Zend\Hydrator\NamingStrategy\UnderscoreNamingStrategy;
use Zend\Hydrator\ObjectProperty as ObjectPropertyHydrator;

$underscoreNamingStrategy = new UnderscoreNamingStrategy();
$namingStrategy = new CompositeNamingStrategy([
    'isPublished' => new MapNamingStrategy(['published' => 'isPublished']),
    'publishedOn' => $underscoreNamingStrategy,
    'updatedOn'   => $underscoreNamingStrategy,
]);

$hydrator = new ObjectPropertyHydrator();
$hydrator->setNamingStrategy($namingStrategy);
$data = $hydrator->extract($transaction);
```

Our data will now have the keys `published` , `published_on` , and `updated_on` !

Unfortunately, if we try and hydrate using our `CompositeNamingStrategy` , we'll run into issues; the `CompositeNamingStrategy` does not know how to map the normalized, extracted property names to those the object accepts because it maps a property name to the naming strategy. So, to fix that, we need to add the reverse keys:

```
$mapNamingStrategy = new MapNamingStrategy(['published' => 'isPublished']);
$underscoreNamingStrategy = new UnderscoreNamingStrategy();
$namingStrategy = new CompositeNamingStrategy([
    // Extraction:
    'isPublished'  => $mapNamingStrategy,
    'publishedOn'  => $underscoreNamingStrategy,
    'updatedOn'    => $underscoreNamingStrategy,

    // Hydration:
    'published'    => $mapNamingStrategy,
    'published_on' => $underscoreNamingStrategy,
    'updated_on'   => $underscoreNamingStrategy,
]);
```

Delegation

Sometimes we want to compose a single hydrator, but don't know until runtime what objects we'll be extracting or hydrating. A great example of this is when using zend-db's `HydratingResultSet` , where the hydrator may vary based on the table from which we pull values. Other times, we may want to use the same basic hydrator type, but compose different filters, strategies, or naming strategies based on the object we wish to hydrate or extract.

To accommodate these scenarios, we have two features. The first is `Zend\Hydrator\HydratorPluginManager`. This is a specialized `Zend\ServiceManager\AbstractPluginManager` for retrieving different hydrator instances. When used in zend-mvc or Expressive applications, it can be configured via the `hydrators` configuration key, which uses the semantics for zend-servicemanager, and maps the service to `HydratorManager`.

As an example, we could have the following configuration:

```
return [
    'hydrators' => [
        'factories' => [
            'Acme\BookHydrator' => \Acme\BookHydratorFactory::class,
            'Acme\AuthorHydrator' => \Acme\AuthorHydratorFactory::class,
        ],
    ],
];
```

> ## Manually configuring the HydratorPluginManager
>
> You can also use the `HydratorPluginManager` programmatically:
>
> ```
> $hydrators = new HydratorPluginManager();
> $hydrators->setFactory('Acme\BookHydrator', \Acme\BookHydratorFactory::class);
> $hydrators->setFactory('Acme\AuthorHydrator', \Acme\AuthorHydratorFactory::class)
> ;
> ```

The factories might create standard hydrator instances, but configure them differently:

```php
namespace Acme;

use Psr\Container\ContainerInterface;
use Zend\Hydrator\ObjectProperty;
use Zend\Hydrator\NamingStrategy\CompositeNamingStrategy;
use Zend\Hydrator\NamingStrategy\UnderscoreNamingStrategy;
use Zend\Hydrator\Strategy\DateTimeFormatterStrategy;

class BookHydratorFactory
{
    public function __invoke(ContainerInterface $container)
    {
        $hydrator = new ObjectProperty();
        $hydrator->addFilter('isbn', function ($property) {
            return $property !== 'isbn';
        });
        $hydrator->setNamingStrategy(new CompositeNamingStrategy([
            'publishedOn' => new UnderscoreNamingStrategy(),
        ]));
        $hydrator->setStrategy(new CompositeNamingStrategy([
            'published_on' => new DateTimeFormatterStrategy(),
        ]));
        return $hydrator;
    }
}

class AuthorHydratorFactory
{
    public function __invoke(ContainerInterface $container)
    {
        $hydrator = new ObjectProperty();
        $hydrator->setNamingStrategy(new UnderscoreNamingStrategy());
        return $hydrator;
    }
}
```

You could then compose the `HydratorManager` service in your own class, and pull these hydrators in order to extract or hydrate instances:

```php
$bookData = $hydrators->get('Acme\BookHydrator')->extract($book);
$authorData = $hydrators->get('Acme\AuthorHydrator')->extract($author);
```

The `DelegatingHydrator` works by composing a `HydratorPluginManager` instance, but has an additional semantic: it uses the *class name of the object it is extracting*, or the object type to hydrate, as the *service name* to pull from the `HydratorPluginManager`. As such, we would change our configuration of the hydrators as follows:

```
return [
    'hydrators' => [
        'factories' => [
            \Acme\Book::class => \Acme\BookHydratorFactory::class,
            \Acme\Author::class => \Acme\AuthorHydratorFactory::class,
        ],
    ],
];
```

Additionally, we need to tell our application about the `DelegatingHydrator`:

```
// zend-mvc applications:
return [
    'service_manager' => [
        'factories' => [
            \Zend\Hydrator\DelegatingHydrator::class => \Zend\Hydrator\DelegatingHydra
torFactory::class
        ]
    ],
];

// Expressive applications
return [
    'dependencies' => [
        'factories' => [
            \Zend\Hydrator\DelegatingHydrator::class => \Zend\Hydrator\DelegatingHydra
torFactory::class
        ]
    ],
];
```

Manually creating the DelegatingHydrator

You can instantiate the `DelegatingHydrator` manually; when you do, you pass it the `HydratorPluginManager instance.

```
use Zend\Hydrator\DelegatingHydrator;
use Zend\Hydrator\HydratorPluginManager;

$hydrators = new HydratorPluginManager();
// ... configure the plugin manager ...
$hydrator = new DelegatingHydrator($hydrators);
```

Technically speaking, the `DelegatingHydrator` can accept any PSR-11[1] container to its constructor.

From there, we can inject the `DelegatingHydrator` into any of our own classes, and use it to extract or hydrate objects:

```
$bookData = $hydrator->extract($book);
$authorData = $hydrator->extract($author);
```

This feature can be quite powerful, as it allows you to create the hydration and extraction "recipes" for all of your objects within their own factories, ensuring that anywhere you need them, they operate exactly the same. It also means that for testing purposes, you can simply mock the `HydratorInterface` (or its parents, `ExtractionInterface` and `HydrationInterface`) instead of composing a concrete instance.

Other features

While we've tried to cover the majority of the functionality zend-hydrator provides in this article, it has a number of other useful features:

- The `AggregateHydrator` allows you to handle complex objects that implement multiple common interfaces and/or have nested instances composed; it even exposes events you can listen to during each of extraction and hydration. You can read more about it in the documentation[2].
- You can write objects that provide and expose their own filters by implementing the `Zend\Hydrator\Filter\FilterProviderInterface` .
- You can hydrate or extract arrays of objects by implementing `Zend\Hydrator\Iterator\HydratingIteratorInterface` .

The component can be seen in use in a number of places: zend-db provides a `HydratingResultSet` that leverage the `HydratorPluginManager` in order to hydrate objects pulled from a database. Apigility uses the feature to extract data for Hypertext Application Language (HAL) payloads. We've even seen developers creating custom ORMs for their application using the feature!

What can zend-hydrator help *you* do today?

Footnotes

> [1]. https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-11-container.md ↩

> [2]. https://docs.zendframework.com/zend-hydrator/aggregate/ ↩

Scrape Screens with zend-dom

by Matthew Weier O'Phinney

Even in this day-and-age of readily available APIs and RSS/Atom feeds, many sites offer none of them. How do you get at the data in those cases? Through the ancient internet art of screen scraping.

The problem then becomes: how do you get at the data you need in a pile of HTML soup? You could use regular expressions or any of the various string functions in PHP. All of these are easily subject to error, though, and often require some convoluted code to get at the data of interest.

Alternately, you could treat the HTML as XML, and use the DOM extension[1], which is typically built-in to PHP. Doing so, however, requires more than a passing familiarity with XPath[2], which is something of a black art.

If you use JavaScript libraries or write CSS fairly often, you may be familiar with CSS selectors, which allow you to target either specific nodes or groups of nodes within an HTML document. These are generally rather intuitive:

```
jQuery('section.slide h2').each(function (node) {
  alert(node.textContent);
});
```

What if you could do that with PHP?

Introducing zend-dom

zend-dom[3] provides CSS selector capabilities for PHP, via the `Zend\Dom\Query` class, including:

- element types (`h2` , `span` , etc.)
- class attributes (`.error` , `.next` , etc.)
- element identifiers (`#nav` , `#main` , etc.)
- arbitrary element attributes (`div[onclick="foo"]`), including word matches (`div[role~="navigation"]`) and substring matches (`div[role*="complement"]`)
- descendents (`div .foo span`)

While it does not implement the full spectrum of CSS selectors, it does provide enough to generally allow you to get at the information you need within a page.

Example: retrieving a navigation list

As an example, let's fetch the navigation list from the `Zend\Dom\Query` documentation page itself:

```php
use Zend\Dom\Query;

$html = file_get_contents('https://docs.zendframework.com/zend-dom/query/');
$query = new Query($html);
$results = $query->execute('ul.bs-sidenav li a');

printf("Received %d results:\n", count($results));
foreach ($results as $result) {
    printf("- [%s](%s)\n", $result->getAttribute('href'), $result->textContent);
}
```

The above queries for `ul.bs-sidenav li a` — in other words, all links within list items of the sidenav unordered list.

When you `execute()` a query, you are returned a `Zend\Dom\NodeList` instance, which decorates a DOMNodeList[4] in order to provide features such as `Countable`, and access to the original query and document. In the example above, we `count()` the results, and then loop over them.

Each item in the list is a DOMNode[5], giving you access to any attributes, the text content, and any child elements. In our case, we access the `href` attribute (the link target), and report the text content (the link text).

The results are:

```
Received 3 results:
- [#querying-html-and-xml-documents](Querying HTML and XML Documents)
- [#theory-of-operation](Theory of Operation)
- [#methods-available](Methods Available)
```

Other uses

Another use case is *testing*. When you have classes that return HTML, or if you want to execute requests and test the generated output, you often don't want to test *exact* contents, but rather look for specific data or fragments within the document.

6 7

We provide these capabilities for zend-mvc[6] applications via the zend-test component[7], which provides a number of CSS selector assertions[8] for use in querying the content returned in your MVC responses. Having these capabilities allows testing for dynamic content as well as static content, providing a number of vectors for ensuring application quality.

Start scraping!

We hope you can appreciate the powerful capabilities of this component! We have used this functionality in a variety of ways, from testing applications to creating feeds based on content differences in web pages, to finding and retrieving image URIs from pages.

Get more information from the zend-dom documentation[9].

Footnotes

[1]. http://php.net/dom ↩

[2]. https://en.wikipedia.org/wiki/XPath ↩

[3]. https://docs.zendframework.com/zend-dom/ ↩

[4]. http://php.net/class.domnodelist ↩

[5]. http://php.net/class.domnode ↩

[6]. https://docs.zendframework.com/zend-mvc/ ↩

[7]. https://docs.zendframework.com/zend-test/ ↩

[8]. https://docs.zendframework.com/zend-test/assertions/#css-selector-assertions ↩

[9]. https://docs.zendframework.com/zend-dom/ ↩

Paginating data collections with zend-paginator

by Enrico Zimuel

zend-paginator[1] is a flexible component for paginating collections of data and presenting that data to users.

Pagination[2] is a standard UI solution to manage the visualization of lists of items, like a list of posts in a blog or a list of products in an online store.

zend-paginator is very popular among Zend Framework developers, and it's often used with zend-view[3], thanks to the pagination control view helper zend-view provides.

It can be used also with other template engines. In this article, I will demonstrate how to use it with Plates[4].

Usage of zend-paginator

The component can be installed via Composer:

```
$ composer require zendframework/zend-paginator
```

To consume the paginator component, we need a collection of items. zend-paginator ships with several different adapters for common collection types:

- *ArrayAdapter*, which works with PHP arrays;
- *Callback*, which allows providing callbacks for obtaining counts of items and lists of items;
- *DbSelect*, to work with a SQL collection (using zend-db[5]);
- *DbTableGateway*, to work with a Table Data Gateway (using the TableGateway feature from zend-db.
- *Iterator*, to work with any `Iterator`[6] instance.

If your collection does not fit one of these adapters, you can create a custom adapter. To do so, you will need to implement `Zend\Paginator\Adapter\AdapterInterface`, which defines two methods:

- `count() : int`
- `getItems(int $offset, int $itemCountPerPage) : array`

Each adapter needs to return the total number of items in the collection, implementing the `count()` method, and a portion (a *page*) of items starting from `$offset` position with a size of `$itemCountPerPage` per page.

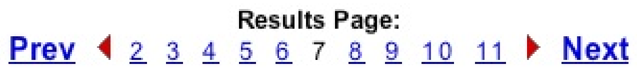
With these two methods, we can use zend-paginator with any type of collection.

For instance, imagine we need to paginate a collection of blog posts and we have a `Posts` class that manages all the posts. We can implement an adapter like this:

```php
require 'vendor/autoload.php';

use Zend\Paginator\Adapter\AdapterInterface;
use Zend\Paginator\Paginator;
use Zend\Paginator\ScrollingStyle\Sliding;

class Posts implements AdapterInterface
{
    private $posts = [];

    public function __construct()
    {
      // Read posts from file/database/whatever
    }

    public function count()
    {
        return count($this->posts);
    }

    public function getItems($offset, $itemCountPerPage)
    {
        return array_slice($this->posts, $offset, $itemCountPerPage);
    }
}

$posts = new Posts();
$paginator = new Paginator($posts);

Paginator::setDefaultScrollingStyle(new Sliding());
$paginator->setCurrentPageNumber(1);
$paginator->setDefaultItemCountPerPage(8);

foreach ($paginator as $post) {
  // Iterate on each post
}

$pages = $paginator->getPages();
var_dump($pages);
```

In this example, we created a zend-paginator adapter using a custom `Posts` class. This class stores the collection of posts using a private array (`$posts`). This adapter is then passed to an instance of `Paginator` .

When creating a `Paginator` , we need to configure its behavior. The first setting is the scrolling style. In the example above, we used the Sliding[7] style, a Yahoo!-like scrolling style that positions the current page number as close as possible to the center of the page range.



> Note: the `Sliding` scrolling style is the default style used by zend-paginator. We need to set it explicitly using `Paginator::setDefaultScrollingStyle()` only if we do not use zend-servicemanager[8] as a plugin manager. Otherwise, the scrolling style is loaded by default from the plugin manager.

The other two configuration values are the current page number and the number of items per page. In the example above, we started from page 1, and we count 8 items per page.

We can then iterate on the `$paginator` object to retrieve the post of the current page in the collection.

At the end, we can retrieve the information regarding the previous page, the next page, the total items in the collection, and more. To get these values we need to call the `getPages()` method. We will obtain an object like this:

```
object(stdClass)#81 (13) {
  ["pageCount"]=>
  int(3)
  ["itemCountPerPage"]=>
  int(8)
  ["first"]=>
  int(1)
  ["current"]=>
  int(1)
  ["last"]=>
  int(3)
  ["next"]=>
  int(2)
  ["pagesInRange"]=>
  array(3) {
    [1]=>
    int(1)
    [2]=>
    int(2)
    [3]=>
    int(3)
  }
  ["firstPageInRange"]=>
  int(1)
  ["lastPageInRange"]=>
  int(3)
  ["currentItemCount"]=>
  int(8)
  ["totalItemCount"]=>
  int(19)
  ["firstItemNumber"]=>
  int(1)
  ["lastItemNumber"]=>
  int(8)
}
```

Using this information, we can easily build an HTML footer to navigate across the collection.

> Note: using zend-view, we can consume the `paginationControl()`[9] helper, which emits an HTML pagination bar.

An example using Plates

Plates[10] implements templates using native PHP; it is fast and easy to use, without any additional meta language; it is just PHP.

In our example, we will create a Plates template to paginate a collection of data using zend-paginator. We will use Bootstrap[11] as the UI framework.

For purposes of this example, blog posts will be accessible via the following URL:

```
/blog[/page/{page:\d+}]
```

where `[/page/{page:\d+}]` represents the optional page number (using the regexp `\d+` to validate only digits). If we open the `/blog` URL we will get the first page of the collection. To return the second page we need to connect to `/blog/page/2`, third page to `/blog/page/3`, and so on.

For instance, we can manage the page parameter using a PSR-7 middleware class consuming the previous `Posts` adapter, that works as follow:

```php
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use League\Plates\Engine;
use Zend\Paginator\Paginator;
use Zend\Paginator\ScrollingStyle\Sliding;
use Posts;

class PaginatorMiddleware
{
    /** @var Posts */
    protected $posts;

    /** @var Engine */
    protected $template;

    public function __construct(Posts $post, Engine $template = null)
    {
        $this->posts    = $post;
        $this->template = $template;
    }

    public function __invoke(
        ServerRequestInterface $request,
        ResponseInterface $response, callable $next = null
    ) {
        $paginator = new Paginator($this->posts);
        $page = $request->getAttribute('page', 1);

        Paginator::setDefaultScrollingStyle(new Sliding());
        $paginator->setCurrentPageNumber($page);
        $paginator->setDefaultItemCountPerPage(8);

        $pages = $paginator->getPages();
        $response->getBody()->write(
            $this->template->render('posts', [
                'paginator' => $paginator,
                'pages'     => $pages,
            ])
        );
        return $response;
    }
}
```

We used a `posts.php` template, passing the paginator (`$paginator`) and the pages (`$pages`) instances. That template could then look like the following:

```
<?php $this->layout('template', ['title' => 'Blog Posts']) ?>

<div class="container">
  <h1>Blog Posts</h1>

  <?php foreach ($paginator as $post) : ?>
    <div class="row">
      <?php // prints the post title, date, author, ... ?>
    </div>
  <?php endforeach ?>

  <?php $this->insert('page-navigation', ['pages' => $pages]) ?>
</div>
```

The `page-navigation.php` template contains the HTML code for the page navigation control, with button like previous, next, and page numbers.

```
<nav aria-label="Page navigation">
  <ul class="pagination">
    <?php if (! isset($pages->previous)) : ?>
      <li class="disabled"><a href="#" aria-label="Previous"><span aria-hidden="true">
&laquo;</span></a></li>
    <?php else : ?>
      <li><a href="/blog/page/<?= $pages->previous ?>" aria-label="Previous"><span ari
a-hidden="true">&laquo;</span></a></li>
    <?php endif ?>

    <?php foreach ($pages->pagesInRange as $num) : ?>
      <?php if ($num === $pages->current) : ?>
        <li class="active"><a href="/blog/page/<?= $num ?>"><?= $num ?> <span class="s
r-only">(current)</span></a></li>
      <?php else : ?>
        <li><a href="/blog/page/<?= $num ?>"><?= $num ?></a></li>
      <?php endif ?>
    <?php endforeach ?>

    <?php if (! isset($pages->next)) : ?>
      <li class="disabled"><a href="#" aria-label="Next"><span aria-hidden="true">&raq
uo;</span></a></li>
    <?php else : ?>
      <li><a href="/blog/page/<?= $pages->next ?>" aria-label="Next"><span aria-hidden
="true">&raquo;</span></a></li>
    <?php endif ?>
  </ul>
</nav>
```

Summary

The zend-paginator component of Zend Framework is a powerful and easy to use package that provides pagination of data. It can be used as standalone component in many PHP projects using different frameworks and template engines. In this article, I demonstrated how to use it in general purpose applications. Moreover, I showed an example using Plates and Bootstrap, in a PSR-7 middleware scenario.

Visit the zend-paginator documentation[12] to find out what else you might be able to do with this component!

Footnotes

> 1. https://docs.zendframework.com/zend-paginator/ ↩

> 2. https://en.wikipedia.org/wiki/Pagination ↩

> 3. https://docs.zendframework.co/zend-view/ ↩

> 4. http://platesphp.com/ ↩

> 5. https://docs.zendframework.com/zend-db/ ↩

> 6. http://php.net/iterator ↩

> 7. https://github.com/zendframework/zend-paginator/blob/master/src/ScrollingStyle/Sliding.php ↩

> 8. https://docs.zendframework.com/zend-servicemanager/ ↩

> 9. https://docs.zendframework.com/zend-paginator/usage/#rendering-pages-with-view-scripts ↩

> 10. http://platesphp.com/ ↩

> 11. http://getbootstrap.com/ ↩

> 12. https://docs.zendframework.com/zend-paginator/ ↩

Logging PHP applications

by Enrico Zimuel

Every PHP application generates errors, warnings, and notices and throws exceptions. If we do not log this information, we lose a way to identify and solve problems at runtime. Moreover, we may need to log specific actions such as a user login and logout attempts. All such information should be filtered and stored in an efficient way.

PHP offers the function error_log()[1] to send an error message to the defined system logger, and the function set_error_handler()[2] to specify a handler for intercepting warnings, errors, and notices generated by PHP.

These functions can be used to customize error management, but it's up to the developer to write the logic to filter and store the data.

Zend Framework offers a logging component, zend-log[3]; the library can be used as a general purpose logging system. It supports multiple log backends, formatting messages sent to the log, and filtering messages from being logged.

Last but not least, zend-log is compliant with PSR-3[4], the logger interface standard.

Installation

You can install zend-log[5] using Composer:

```
composer require zendframework/zend-log
```

Usage

zend-log can be used to create log entries in different formats using multiple backends. You can also filter the log data from being saved, and process the log event prior to filtering or writing, allowing the ability to substitute, add, remove, or modify the data you log.

Basic usage of zend-log requires both a *writer* and a *logger* instance. A writer stores the log entry into a backend, and the logger consumes the writer to perform logging operations.

As an example:

```
use Zend\Log\Logger;
use Zend\Log\Writer\Stream;

$logger = new Logger;
$writer = new Stream('php://output');

$logger->addWriter($writer);
$logger->log(Logger::INFO, 'Informational message');
```

The above produces the following output:

```
2017-09-11T15:07:46+02:00 INFO (6): Informational message
```

The output is a string containing a timestamp, a priority (`INFO (6)`) and the message (`Informational message`). The output format can be changed using the `setFormatter()` method of the writer object (`$writer`). The default log format, produced by the Simple[6] formatter, is as follows:

```
%timestamp% %priorityName% (%priority%): %message% %extra%
```

where `%extra%` is an optional value containing additional information.

For instance, if you wanted to change the format to include only `log %message%` , you could do the following:

```
$formatter = new Zend\Log\Formatter\Simple('log %message%' . PHP_EOL);
$writer->setFormatter($formatter);
```

Log PHP events

zend-log can also be used to log PHP errors and exceptions. You can log PHP errors using the static method `Logger::registerErrorHandler($logger)` and intercept exceptions using the static method `Logger::registerExceptionHandler($logger)` .

```
use Zend\Log\Logger;
use Zend\Log\Writer;

$logger = new Logger;
$writer = new Writer\Stream(__DIR__ . '/test.log');
$logger->addWriter($writer);

// Log PHP errors
Logger::registerErrorHandler($logger);

// Log exceptions
Logger::registerExceptionHandler($logger);
```

Filtering data

As mentioned, we can filter the data to be logged; filtering *removes* messages that match the filter criteria, preventing them from being logged.

We can use the `addFilter()` method of the Writer interface[7] to add a specific filter.

For instance, we can filter by priority, accepting only log entries with a priority less than or equal to a specific value:

```
$filter = new Zend\Log\Filter\Priority(Logger::CRIT);
$writer->addFilter($filter);
```

In the above example, the logger will only store log entries with a priority less than or equal to `Logger::CRIT` (critical). The priorities are defined by the `Zend\Log\Logger` class:

```
const EMERG   = 0;  // Emergency: system is unusable
const ALERT   = 1;  // Alert: action must be taken immediately
const CRIT    = 2;  // Critical: critical conditions
const ERR     = 3;  // Error: error conditions
const WARN    = 4;  // Warning: warning conditions
const NOTICE  = 5;  // Notice: normal but significant condition
const INFO    = 6;  // Informational: informational messages
const DEBUG   = 7;  // Debug: debug messages
```

As such, only emergency, alerts, or critical entries would be logged.

We can also filter log data based on regular expressions, timestamps, and more. One powerful filter uses a zend-validator[8] `ValidatorInterface` instance to filter the log; only valid entries would be logged in such cases.

Processing data

If you need to provide additional information to logs in an automated fashion, you can use a `Zend\Log\Processer` class. A processor is executed before the log data are passed to the writer. The input of a processor is a *log event*, an array containing all of the information to log; the output is also a *log event*, but can contain modified or additional values. A processor modifies the log event to prior to sending it to the writer.

You can read about processor adapters offered by zend-log in the documentation[9].

Multiple backends

One of the cool feature of zend-log is the possibility to write logs using multiple backends. For instance, you can write a log to both a file and a database using the following code:

```php
use Zend\Db\Adapter\Adapter as DbAdapter;
use Zend\Log\Formatter;
use Zend\Log\Writer;
use Zend\Log\Logger;

// Create our adapter
$db = new DbAdapter([
    'driver'   => 'Pdo',
    'dsn'      => 'mysql:dbname=testlog;host=localhost',
    'username' => 'root',
    'password' => 'password'
]);

// Map event data to database columns
$mapping = [
    'timestamp' => 'date',
    'priority'  => 'type',
    'message'   => 'event',
];

// Create our database log writer
$writerDb = new Writer\Db($db, 'log', $mapping); // log table
$formatter = new Formatter\Base();
$formatter->setDateTimeFormat('Y-m-d H:i:s'); // MySQL DATETIME format
$writerDb->setFormatter($formatter);

// Create our file log writer
$writerFile = new Writer\Stream(__DIR__ . '/test.log');

// Create our logger and register both writers
$logger = new Logger();
$logger->addWriter($writerDb, 1);
$logger->addWriter($writerFile, 100);

// Log an information message
$logger->info('Informational message');
```

The database writer requires the credentials to access the table where you will store log information. You can customize the field names for the database table using a `$mapping` array, containing an associative array mapping log fields to database columns.

The database writer is composed in `$writerDb` and the file writer in `$writerFile`. The writers are added to the logger using the `addWriter()` method with a priority number; higher integer values indicate higher priority (triggered earliest). We chose priority 1 for the database writer, and priority 100 for the file writer; this means the file writer will log first, followed by logging to the database.

> Note: we used a special date formatter for the database writer. This is required to translate the log timestamp into the DATETIME format of MySQL.

PSR-3 support

If you need to be compatible with PSR-3[10], you can use `Zend\Log\PsrLoggerAdapter`. This logger can be used anywhere a `Psr\Log\LoggerInterface` is expected.

As an example:

```
use Psr\Log\LogLevel;
use Zend\Log\Logger;
use Zend\Log\PsrLoggerAdapter;

$zendLogLogger = new Logger;
$psrLogger = new PsrLoggerAdapter($zendLogLogger);

$psrLogger->log(LogLevel::INFO, 'We have a PSR-compatible logger');
```

To select a PSR-3 backend for writing, we can use the `Zend\Log\Writer\Psr` class. In order to use it, you need to pass a `Psr\Log\LoggerInterface` instance to the `$psrLogger` constructor argument:

```
$writer = new Zend\Log\Writer\Psr($psrLogger);
```

zend-log also supports PSR-3 message placeholders[11] via the `Zend\Log\Processor\PsrPlaceholder` class. To use it, you need to add a `PsrPlaceholder` instance to a logger, using the `addProcess()` method. Placeholder names correspond to keys in the "extra" array passed when logging a message:

```
use Zend\Log\Logger;
use Zend\Log\Processor\PsrPlaceholder;

$logger = new Logger;
$logger->addProcessor(new PsrPlaceholder);

$logger->info('User with email {email} registered', ['email' => 'user@example.org']);
```

An informational log entry will be stored with the message `User with email user@example.org registered`.

Logging an MVC application

If you are using a zend-mvc[12] based application, you can use zend-log as module. zend-log provides a Module.php[13] class, which registers `Zend\Log` as a module in your application.

In particular, the zend-log module provides the following services (under the namespace `Zend\Log`):

```
Logger::class          => LoggerServiceFactory::class,
'LogFilterManager'     => FilterPluginManagerFactory::class,
'LogFormatterManager' => FormatterPluginManagerFactory::class,
'LogProcessorManager' => ProcessorPluginManagerFactory::class,
'LogWriterManager'     => WriterPluginManagerFactory::class,
```

The `Logger::class` service can be configured using the `log` config key; the documentation provides configuration examples[14].

In order to use the `Logger` service in your MVC stack, grab it from the service container. For instance, you can pass the *Logger service* in a controller using a factory:

```
use Zend\Log\Logger;
use Zend\ServiceManager\Factory\FactoryInterface;

class IndexControllerFactory implements FactoryInterface
{
    public function __invoke(
        ContainerInterface $container,
        $requestedName,
        array $options = null
    ) {
        return new IndexController(
            $container->get(Logger::class)
        );
    }
}
```

via the following service configuration for the `IndexController`:

```
'controllers' => [
    'factories' => [
        IndexController::class => IndexControllerFactory::class,
    ],
],
```

Logging a middleware application

You can also integrate zend-log in your middleware applications. If you are using Expressive[15], add the component's ConfigProvider[16] to your `config/config.php` file.

17

> Note: if you are using zend-component-installer[17], you will be prompted to install zend-log's config provider when you install the component via Composer.
>
> Note: This configuration registers the same services provided in the zend-mvc example, above.

To use zend-log in middleware, grab it from the dependency injection container and pass it as a dependency to your middleware:

```
namespace App\Action;

use Psr\Container\ContainerInterface;
use Zend\Log\Logger;

class HomeActionFactory
{
    public function __invoke(ContainerInterface $container) : HomeAction
    {
        return new HomeAction(
            $container->get(Logger::class)
        );
    }
}
```

As an example of logging in middleware:

```php
namespace App\Action;

use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface as ServerMiddlewareInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Log\Logger;

class HomeAction implements ServerMiddlewareInterface
{
    private $logger;

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

    public function process(
        ServerRequestInterface $request,
        DelegateInterface $delegate
    ) {
        $this->logger->info(__CLASS__ . ' has been executed');

        // ...
    }
}
```

Listening for errors in Expressive

Expressive and Stratigility[18] provide a default error handler middleware implementation, `Zend\Stratigility\Middleware\ErrorHandler` which listens for PHP errors and exceptions/throwables. By default, it spits out a simple error page when an error occurs, but it also provides the ability to attach *listeners*, which can then act on the provided error.

Listeners receive the error, the request, and the response that the error handler will be returning. We can use that information to log information!

First, we create an error handler listener that composes a logger, and logs the information:

```php
use Exception;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Throwable;
use Zend\Log\Logger;

class LoggingErrorListener
{
    /**
     * Log message string with placeholders
     */
    const LOG_STRING = '{status} [{method}] {uri}: {error}';

    private $logger;

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

    public function __invoke(
        $error,
        ServerRequestInterface $request,
        ResponseInterface $response
    ) {
        $this->logger->error(self::LOG_STRING, [
            'status' => $response->getStatusCode(),
            'method' => $request->getMethod(),
            'uri'    => (string) $request->getUri(),
            'error'  => $error->getMessage(),
        ]);
    }
}
```

> The `ErrorHandler` implementation casts PHP errors to `ErrorException` instances, which means that `$error` is always some form of throwable.

We can then write a delegator factory that will register this as a listener on the `ErrorHandler`:

```php
use LoggingErrorListener;
use Psr\Container\ContainerInterface;
use Zend\Log\Logger;
use Zend\Log\Processor\PsrPlaceholder;
use Zend\Stratigility\Middleware\ErrorHandler;

class LoggingErrorListenerFactory
{
    public function __invoke(
        ContainerInterface $container,
        $serviceName,
        callable $callback
    ) : ErrorHandler {
        $logger = $container->get(Logger::class);
        $logger->addProcessor(new PsrPlaceholder());

        $listener = new LoggingErrorListener($logger);

        $errorHandler = $callback();
        $errorHandler->attachListener($listener);
        return $errorHandler;
    }
}
```

And then register the delegator in your configuration:

```php
// In a ConfigProvider, or a config/autoload/*.global.php file:
use LoggingErrorListenerFactory;
use Zend\Stratigility\Middleware\ErrorHandler;

return [
    'dependencies' => [
        'delegators' => [
            ErrorHandler::class => [
                LoggingErrorListenerFactory::class,
            ],
        ],
    ],
];
```

At this point, your error handler will now also log errors to your configured writers!

Summary

The zend-log component offers a wide set of features, including support for multiple writers, filtering of log data, compatibility with PSR-3[19], and more.

Hopefully you can use the examples above for consuming zend-log in your standalone, zend-mvc, Expressive, or general middleware applications!

Learn more in the zend-log documentation[20].

Footnotes

> [1]. http://php.net/error_log ↩

> [2]. http://php.net/set_error_handler ↩

> [3]. https://docs.zendframework.com/zend-log/ ↩

> [4]. http://www.php-fig.org/psr/psr-3/ ↩

> [5]. https://docs.zendframework.com/zend-log/ ↩

> [6]. https://github.com/zendframework/zend-log/blob/master/src/Formatter/Simple.php ↩

> [7]. https://github.com/zendframework/zend-log/blob/master/src/Writer/WriterInterface.php ↩

> [8]. https://docs.zendframework.com/zend-validator/ ↩

> [9]. https://docs.zendframework.com/zend-log/processors/ ↩

> [10]. http://www.php-fig.org/psr/psr-3/ ↩

> [11]. http://www.php-fig.org/psr/psr-3/#12-message ↩

> [12]. https://docs.zendframework.com/zend-mvc/ ↩

> [13]. https://github.com/zendframework/zend-log/blob/master/src/Module.php ↩

> [14]. https://docs.zendframework.com/zend-log/service-manager/#zend-log-as-a-module ↩

> [15]. https://docs.zendframework.com/zend-expressive/ ↩

> [16]. https://github.com/zendframework/zend-log/blob/master/src/ConfigProvider.php ↩

> [17]. https://docs.zendframework.com/zend-component-installer/ ↩

> [18]. https://docs.zendframework.com/zend-stratigility/ ↩

> [19]. http://www.php-fig.org/psr/psr-3/ ↩

> [20]. https://docs.zendframework.com/zend-log/ ↩

Discover and Read RSS and Atom Feeds

by Matthew Weier O'Phinney

Remember RSS and Atom feeds?

Chances are, you may have discovered this book *because* it was announced on a feed:

- A number of Twitter services poll feeds and send links when new entries are discovered.
- Some of you may be using feed readers such as Feedly[1].
- Many news aggregator services, including tools such as Google Now, use RSS and Atom feeds as sources.

An interesting fact: Atom itself is often used as a data transfer format for REST services, particularly content management platforms! As such, being familiar with feeds and having tools to work with them is an important skill for a web developer!

In this first of a two part series on feeds, we'll look at feed *discovery*, as well as *reading*, using zend-feed's Reader subcomponent.

Getting started

First, of course, you need to install zend-feed:

```
$ composer require zendframework/zend-feed
```

As of version 2.6.0, the component has a very minimal set of dependencies: it only requires zendframework/zend-escaper and zendframework/zend-stdlib in order to work. It has a number of additional, optional requirements depending on features you want to opt-in to:

- psr/http-message and/or zend-http, to allow polling pages for feeds, feeds themselves, or PubSubHubbub services.
- zendframework/zend-cache, to allow caching feeds between requests.
- zendframework/zend-db, which is used when using the PubSubHubbub subcomponent, in order for PuSH subscribers to store updates.
- zendframework/zend-validator, for validating addresses used in Atom feeds and entries when using the Writer subcomponent.

For our examples, we will need an HTTP client in order to fetch pages. For the sake of simplicity, we'll go ahead and use zendframework/zend-http; if you are already using Guzzle in your application, you can create a wrapper for it following instructions in the zend-feed manual[2].

```
$ composer require zendframework/zend-http
```

Now that we have these pieces in place, we can move on to link discovery!

Link discovery

The Reader subcomponent contains facilities for finding Atom and RSS links within an HTML page. Let's try this now:

```
// In discovery.php:

use Zend\Feed\Reader\Reader;

require 'vendor/autoload.php';

$feedUrls  = [];
$feedLinks = Reader::findFeedLinks('https://framework.zend.com');

foreach ($feedLinks as $link) {
    switch ($link['type']) {
        case 'application/atom+xml':
            $feedUrls[] = $link['href'];
            break;
        case 'application/rss+xml':
            $feedUrls[] = $link['href'];
            break;
    }
}

var_export($feedUrls);
```

If you run the above, you should get a list like the following (at the time of writing):

```
array (
  0 => 'https://framework.zend.com/security/feed',
  1 => 'https://framework.zend.com/blog/feed-atom.xml',
  2 => 'https://framework.zend.com/blog/feed-rss.xml',
  3 => 'https://framework.zend.com/releases/atom.xml',
  4 => 'https://framework.zend.com/releases/rss.xml',
)
```

That's rather useful! We can poll a page to discover links, and then follow them!

Internally, the returned `$feedLinks` is a `Zend\Feed\Reader\FeedSet` instance, which is really just an `ArrayObject` where each item it composes is itself a `FeedSet` with specific attributes set (including the `type` , `href` , and `rel` , usually). It only returns links that are known feed types; any other type of link is ignored.

Reading a feed

Now that we know where some feeds are, we can read them.

To do that, we pass a URL for a feed to the reader, and then pull data from the returned feed:

```
// In reader.php:

use Zend\Feed\Reader\Reader;

require 'vendor/autoload.php';

$feed = Reader::import('https://framework.zend.com/releases/rss.xml');

printf(
    "[%s](%s): %s\n",
    $feed->getTitle(),
    $feed->getLink(),
    $feed->getDescription()
);
```

The above will result in:

```
[Zend Framework Releases](https://github.com/zendframework): Zend Framework and zfcamp
us releases
```

The above is considered the feed *channel data*; it's information about the feed itself. Most likely, though, we want to know what *entries* are in the feed!

Getting feed entries

The feed returned by `Reader::import()` is itself *iterable*, which each item of iteration being an *entry*. At its most basic:

```
foreach ($feed as $entry) {
    printf(
        "[%s](%s): %s\n",
        $entry->getTitle(),
        $entry->getLink(),
        $entry->getDescription()
    );
}
```

This will loop through each entry, listing the title, the canonical link to the item, and a description of the entry.

The above will work across any type of feed. However, feed capabilities vary based on type. RSS and Atom feed entries will have different data available; in fact, Atom is considered an *extensible* protocol, which means that such entries can potentially expose quite a lot of additional data!

You may want to read up on what's available; follow the footnotes to find relevant links:

- RSS entry properties[3]
- Atom entries[4]

Until next time

zend-feed's Reader subcomponent offers a number of other capabilities, including:

- Importing actual feed *strings* (versus fetching via an HTTP client)
- The ability to utilize alternate HTTP clients.
- The ability to extend the Atom protocol in order to access additional data.

The zend-feed component has extensive documentation[5], which will answer most questions you may have at this point.

We hope this quick primer gets you started consuming feeds!

Footnotes

1. https://feedly.com ↩

2. https://docs.zendframework.com/zend-feed/psr7-clients/ ↩

3. https://docs.zendframework.com/zend-feed/consuming-rss/#get-properties ↩

4. https://docs.zendframework.com/zend-feed/consuming-atom/ ↩

5

> 5. https://docs.zendframework.com/zend-feed/ ↩

Create RSS and Atom Feeds

by Matthew Weier O'Phinney

In the previous article, we detailed RSS and Atom feed discovery and parsing. In this article we're going to cover its complement: feed creation!

zend-feed provides the ability to create both Atom 1.0 and RSS 2.0 feeds, and even supports custom extensions during feed generation, including:

- Atom (`xmlns:atom` ; RSS 2 only): provide links to Atom feeds and Pubsubhubbub URIs within your RSS feed.
- Content (`xmlns:content` ; RSS 2 only): provide CDATA encoded content for individual feed items.
- DublinCore (`xmlns:dc` ; RSS 2 only): provide metadata around common content elements such as author/publisher/contributor/creator, dates, languages, etc.
- iTunes (`xmlns:itunes`): create podcast feeds and items compatible with iTunes.
- Slash (`xmlns:slash` ; RSS 2 only): communicate comment counts per item.
- Threading (`xmlns:thr` ; RSS 2 only): provide metadata around *threading* feed items, including indicating what an item is *in reply to*, linking to *replies*, and metrics around each.
- WellFormedWeb (`xmlns:wfw` ; RSS 2 only): provide a link to a separate comments feed for a given entry.

You can also provide your own custom extensions if desired; these are just what we ship out of the box! In many cases, you don't even need to know about the extensions, as zend-feed will take care of adding in those that are required, based on the data you provide in the feed and entries.

Creating a feed

The first step, of course, is having some content! I'll assume you have items you want to publish, and those will be in `$data` , which we'll loop over. How that data looks will be dependent on your application, so please be aware that you may need to adjust any examples below to fit your own data source.

Next, we need to have zend-feed installed; do that via Composer:

```
$ composer require zendframework/zend-feed
```

Now we can finally get started. We'll begin by creating a feed, and populating it with some basic metadata:

```php
use Zend\Feed\Writer\Feed;

$feed = new Feed();
// Title of the feed
$feed->setTitle('Tutorial Feed');
// Link to the feed's target, usually a homepage:
$feed->setLink('https://example.com/');
// Link to the feed itself, and the feed type:
$feed->setFeedLink('https://example.com/feed.xml', 'rss');

// Feed description; only required for RSS:
$feed->setDescription('This is a tutorial feed for example.com');
```

A couple things to note: First, you need to know what *type* of feed you're creating up front, as it will affect what properties *must* be set, as well as which are actually available. I personally like to generate feeds of both types, so I'll do the above within a method call that accepts the feed type as an argument, and then puts some declarations within conditionals based on that type.

Second, you'll need to know the fully-qualified URIs to the feed target and the feed itself. These will generally be something you generate; most routing libraries will have these capabilities, and you'll generate these within your application, instead of hard-coding them as I have done here.

Adding items

Now that we have our feed, we'll loop over our data set and add items. Items generally have:

- a title
- a link to the item
- an author
- the dates when it was modified, and last updated
- content

Putting it together:

```
$latest = new DateTime('@0');
foreach ($data as $datum) {
    // Create an empty entry:
    $entry = $feed->createEntry();

    // Set the entry title:
    $entry->setTitle($datum->getTitle());

    // Set the link to the entry:
    $entry->setLink(sprintf('%s%s.html', $baseUri, $datum->getId()));

    // Add an author, if you can. Each author entry should be an
    // array containing minimally a "name" key, and zero or more of
    // the keys "email" or "uri".
    $entry->addAuthor($datum->getAuthor());

    // Set the date created:
    $entry->setDateCreated(new DateTime($datum->getDateCreated()));

    // And the date last updated:
    $modified = new DateTime($datum->getDateModified());
    $entry->setDateModified($modified);

    // And finally, some content:
    $entry->setContent($datum->getContent());

    // Add the new entry to the feed:
    $feed->addEntry($entry);

    // And memoize the date modified, if it's more recent:
    $latest = $modified > $latest ? $modified : $latest;
}
```

There are quite a few other properties you can set, and some of these will vary based on custom extensions you might register with the feed; the above are the typical items you'll include in a feed entry, however.

What is that bit about `$latest`, though?

Feeds need to have a timestamp indicating when they were most recently modified.

Why? Because feeds are intended to be read by *machines* and aggregators, and need to know when *new* content is available.

You could set the date of modification to whatever the current timestamp is at time of execution, but it's better to have it in sync with the most recent entry in the feed itself. As such, the above code creates a timestamp set to timestamp `0`, and checks for a modified date that is newer on each iteration.

Once we have that in place, we can add the modified date to the feed itself:

```
$feed->setDateModified($latest);
```

Rendering the feed

Rendering the feed involves *exporting* it, which requires knowing the feed *type*; this is necessary so that the correct XML markup is generated.

So, let's create an RSS feed:

```
$rss = $feed->export('rss');
```

If we wanted, and we have the correct properties present, we can also render Atom:

```
$atom = $feed->export('atom');
```

Now what?

I often pre-generate feeds and cache them to the filesystem. In that case, a `file_put_contents()` call, using the generated feed as the string contents, is all that's needed.

If you're serving the feed back over HTTP, you will want to send back the correct HTTP `Content-Type` when you do. Additionally, it's good to send back a `Last-Modified` header with the same date as the feed's own last modified date, and/or an ETag with a hash of the feed; these allow clients performing HEAD requests to determine whether or not they need to retrieve the full content, or if they already have the latest.

If you are using PSR-7 middleware, these processes might look like this:

```php
use Zend\Diactoros\Response\TextResponse;

$commonHeaders = [
    'Last-Modified' => $feed->getDateModified()->format('c'),
    'ETag' => hash('sha256', $feed)
];

// For an RSS feed:
return new TextResponse($rss, 200, array_merge(
    $commonHeaders,
    ['Content-Type' => 'application/rss+xml']
));

// For an Atom feed:
return new TextResponse($atom, 200, array_merge(
    $commonHeaders,
    ['Content-Type' => 'application/atom+xml']
));
```

Summing up

zend-feed's generation capabilities are incredibly flexible, while making the general use-case straight-forward. We have created feeds for blog posts, releases, tweets, and commenting systems using the component; it does exactly what it advertises.

Visit the zend-feed documentation[1] for more information.

Footnotes

1. https://docs.zendframework.com/zend-feed/ ↩

Manage permissions with zend-permissions-rbac

by Matthew Weier O'Phinney

In the article Manage permissions with zend-permissions-acl, we cover usage of Access Control Lists (ACL) for managing user permissions. In this article, we'll cover another option provided by Zend Framework, zend-permissions-rbac[1], our lightweight role-based access control (RBAC) implementation.

Installing zend-permissions-rbac

Just as you would any of our components, install zend-permissions-rbac via Composer:

```
$ composer require zendframework/zend-permissions-rbac
```

The component has no requirements at this time other than a PHP version of at least 5.5.

Vocabulary

In RBAC systems, we have three primary items to track:

- The **RBAC** system composes zero or more *roles*.
- A **role** is granted zero or more *permissions*.
- We **assert** whether or not a *role* is *granted* a given *permission*.

zend-permissions-rbac supports role inheritance, even allowing a role to inherit permissions from multiple other roles. This allows you to create some fairly complex and fine-grained permissions schemes!

Basics

As a basic example, we'll create an **RBAC** for a content-based website. Let's start with a "guest" **role**, that only allows "read" permissions.

```
use Zend\Permissions\Rbac\Rbac;
use Zend\Permissions\Rbac\Role;

// Create some roles
$guest= new Role('guest');
$guest->addPermission('read');

$rbac = new Rbac();
$rbac->addRole($guest);
```

We can then **assert** if a given role *is granted* specific permissions:

```
$rbac->isGranted('guest', 'read'); // true
$rbac->isGranted('guest', 'write'); // false
```

Unknown roles

One thing to note: if the role used with `isGranted()` does not exist, this method *raises an exception*, specifically a `Zend\Permissions\Rbac\Exception\InvalidArgumentException`, indicating the role could not be found.

In many situations, this may not be what you want; you may want to handle non-existent roles gracefully. You could do this in three ways.

First, you can test to see if the role exists *before* you check the permissions, using `hasRole()`:

```
if (! $rbac->hasRole($foo)) {
    // failed, due to missing role
}
if (! $rbac->isGranted($foo, $permission)) {
    // failed, due to missing permissions
}
```

Second, you can wrap the `isGranted()` call in a try/catch block:

> ```
> try {
>     if (! $rbac->isGranted($foo, $permission)) {
>         // failed, due to missing permissions
>     }
> } catch (RbacInvalidArgumentException $e) {
>     if (! strstr($e->getMessage(), 'could be found')) {
>         // failed, due to missing role
>     }
>
>     // some other error occured
>     throw $e;
> }
> ```
>
> Personally, I don't like to use exceptions for application flow; that said, in most cases, you will be working with a role instance that you've just added to the RBAC.
>
> Third, zend-permissions-rbac has a built-in mechanism for this:
>
> ```
> $rbac->setCreateMissingRoles(true);
> ```
>
> After calling this method, any `isGranted()` calls you make with unknown role identifiers will simply return a boolean `false`.
>
> This method also ensures you do not encounter errors when creating role inheritance chains and add roles out-of-order (e.g., adding children which have not yet been created to a role you are defining).
>
> As such, please assume that all further examples have called this method if creation of the RBAC instance is not demonstrated.

Role inheritance

Let's say we want to build on the previous example, and create an "editor" role that also incorporates the permissions of the "guest" role, and adds a "write" permission.

You might be inclined to think of the "editor" as *inheriting* from the "guest" role — in other words, that it is a *descendent* or *child* of it. However, in RBAC, inheritance works in the opposite direction: a *parent* inherits all permissions of its *children*. As such, we'll create the role as follows:

```php
$editor = new Role('editor');
$editor->addChild($guest);
$editor->addPermission('write');

$rbac->addRole($editor);

$rbac->isGranted('editor', 'write'); // true
$rbac->isGranted('editor', 'read');  // true
$rbac->isGranted('guest',  'write'); // false
```

Another role might be a "reviewer" who can "moderate" content:

```php
$reviewer = new Role('reviewer');
$reviewer->addChild($guest);
$reviewer->addPermission('moderate');

$rbac->addRole($reviewer);

$rbac->isGranted('reviewer', 'moderate'); // true
$rbac->isGranted('reviewer', 'write');    // false; editor only!
$rbac->isGranted('reviewer', 'read');     // true
$rbac->isGranted('guest',    'moderate'); // false
```

Let's create another, an "admin" who can do **all** of the above, but also has permissions for "settings":

```php
$admin= new Role('admin');
$admin->addChild($editor);
$admin->addChild($reviewer);
$admin->addPermission('settings');

$rbac->addRole($admin);

$rbac->isGranted('admin',    'settings'); // true
$rbac->isGranted('admin',    'write');    // true
$rbac->isGranted('admin',    'moderate'); // true
$rbac->isGranted('admin',    'read');     // true
$rbac->isGranted('editor',   'settings'); // false
$rbac->isGranted('reviewer', 'settings'); // false
$rbac->isGranted('guest',    'write');    // false
```

As you can see, permissions lookups are *recursive* and *collective*; the RBAC examines *all* children and each of their descendants as far down as it needs to determine if a given permission is granted!

Creating your RBAC

When should you create your RBAC, exactly? And should it contain all roles and permissions?

In most cases, you will be validating a single user's permissions. What's interesting about zend-permissions-rbac is that if you know that user's role, the permissions they have been assigned, and any child roles (and their permissions) to which the role belongs, you have everything you need. This means that you can do most lookups on-the-fly.

As such, you will typically do the following:

- Create a finite set of well-known roles and their permissions as a global RBAC.
- Add roles (and optionally permissions) for the current user.
- Validate the current user against the RBAC.

As an example, let's say I have a user Mario who has the role "editor", and also adds the permission "update". If our RBAC is already populated per the above examples, I might do the following:

```
$mario= new Role('mario');
$mario->addChild($editor);
$mario->addPermission('update');

$rbac->addRole($mario);

$rbac->isGranted($mario,   'settings'); // false; admin only!
$rbac->isGranted($mario,   'update');   // true; mario only!
$rbac->isGranted('editor', 'update');   // false; mario only!
$rbac->isGranted($mario,   'write');    // true; all editors
$rbac->isGranted($mario,   'read');     // true; all guests
```

Assigning roles to users

When you have some sort of authentication system in place, it will return some sort of *identity* or *user* instance generally. You will then need to map this to RBAC roles. But how?

Hopefully, you can store role information wherever you persist your user information. Since roles are essentially stored internally as strings by zend-permissions-rbac, this means that you can store the user role as a discrete datum with your user identity.

Once you have, you have a few options:

- Use the role directly from your identity when checking permissions: e.g., `$rbac->isGranted($identity->getRole(), 'write')`
- Create a `Zend\Permissions\Rbac\Role` instance (or other concrete class) with the role fetched from the identity, and use that for permissions checks: `$rbac->isGranted(new`

`Role($identity->getRole()), 'write')`

- Update your identity instance to implement `Zend\Permissions\Rbac\RoleInterface`, and pass it directly to permissions checks: `$rbac->isGranted($identity, 'write')`

This latter approach provides a nice solution, as it then also allows you to store specific *permissions* and/or *child roles* as part of the user data.

The `RoleInterface` looks like the following:

```
namespace Zend\Permissions\Rbac;

use RecursiveIterator;

interface RoleInterface extends RecursiveIterator
{
    /**
     * Get the name of the role.
     *
     * @return string
     */
    public function getName();

    /**
     * Add permission to the role.
     *
     * @param $name
     * @return RoleInterface
     */
    public function addPermission($name);

    /**
     * Checks if a permission exists for this role or any child roles.
     *
     * @param  string $name
     * @return bool
     */
    public function hasPermission($name);

    /**
     * Add a child.
     *
     * @param  RoleInterface|string $child
     * @return Role
     */
    public function addChild($child);

    /**
     * @param  RoleInterface $parent
     * @return RoleInterface
     */
    public function setParent($parent);

    /**
     * @return null|RoleInterface
     */
    public function getParent();
}
```

The `Zend\Permissions\Rbac\AbstractRole` contains basic implementations of most methods of the interface, including logic for querying child permissions, so we suggest inheriting from that if you can.

As an example, you could store the permissions as a comma-separated string and the parent role as a string internally when creating your identity instance:

```php
use Zend\Permissions\Rbac\AbstractRole;
use Zend\Permissions\Rbac\RoleInterface;
use Zend\Permissions\Rbac\Role;

class Identity extends AbstractRole
{
    /**
     * @param string $username
     * @param string $role
     * @param array $permissions
     * @param array $childRoles
     */
    public function __construct(
        string $username,
        array $permissions = [],
        array $childRoles = []
    ) {
        // $name is defined in AbstractRole
        $this->name = $username;

        foreach ($this->permissions as $permission) {
            $this->addPermission($permission);
        }

        $childRoles = array_merge(['guest'], $childRoles);
        foreach ($this->childRoles as $childRole) {
            $this->addChild($childRole);
        }
    }
}
```

Assuming your authentication system uses a database table, and a lookup returns an array-like row with the user information on a successful lookup, you might then seed your identity instance as follows:

```php
$identity = new Identity(
    $row['username'],
    explode(',', $row['permissions']),
    explode(',', $row['roles'])
);
```

This approach allows you to assign pre-determined roles to individual users, while also allowing you to add fine-grained, individual permissions!

Custom assertions

Sometimes a static assertion is not enough.

As an example, we may want to implement a rule that the *creator* of a content item in our website always has rights to *edit* the item. How would we implement that with the above system?

zend-permissions-rbac allows you to do so via dynamic assertions. Such assertions are classes that implement `Zend\Permissions\Rbac\AssertionInterface`, which defines the single method `public function assert(Rbac $rbac)`.

For the sake of this example, let's assume:

- The content item is represented as an object.
- The object has a method `getCreatorUsername()` that will return the same username as we might have in our custom identity from the previous example.

Because we have PHP 7 at our disposal, we'll create the assertion as an anonymous class:

```
use Zend\Permissions\Rbac\AssertionInterface;
use Zend\Permissions\Rbac\Rbac;
use Zend\Permissions\Rbac\RoleInterface;

$assertion = new class ($identity, $content) implements AssertionInterface {
    private $content;
    private $identity;

    public function __construct(RoleInterface $identity, $content)
    {
        $this->identity = $identity;
        $this->content = $content;
    }

    public function assert(Rbac $rbac)
    {
        return $this->identity->getName() === $this->content->getCreatorUsername();
    }
};

$rbac->isGranted($mario, 'edit', $assertion); // returns true if $mario created $conte
nt
```

This opens even more possibilities than inheritance!

Summary

zend-permissions-rbac is quite simple to operate, but that simplicity hides a great amount of flexibility and power; you can create incredibly fine-grained permissions schemes for your applications using this component!

Footnotes

> [1]. https://docs.zendframework.com/zend-permissions-rbac/ ↩

Manage permissions with zend-permissions-acl

by Matthew Weier O'Phinney

In the article Manage permissions with zend-permissions-rbac, we cover usage of Role Based Access Controls (RBAC). In this article, we'll explore another option provided by Zend Framework, zend-permissions-acl[1], which implements Access Control Lists (ACL).

This post will follow the same basic format as the one covering zend-permissions-rbac, using the same basic examples.

Installing zend-permissions-acl

Just as you would any of our components, install zend-permissions-acl via Composer:

```
$ composer require zendframework/zend-permissions-acl
```

The component has no requirements at this time other than a PHP version of at least 5.5.

Vocabulary

In ACL systems, we have three concepts:

- a **resource** is something to which we control access.
- a **role** is something that will request access to a *resource*.
- Each *resource* has **privileges** for which access will be requested to specific *roles*.

As an example, an *author* might request to *create* (privilege) a *blog post* (resource); later, an *editor* (role) might request to *edit* (privilege) a *blog post* (resource).

The chief difference to RBAC is that RBAC essentially combines the resource and privilege into a single item. By separating them, you can create a set of discrete permissions for your *entire* application, and then create roles with multiple-inheritance in order to implement fine-grained permissions.

ACLs

An ACL is created by instantiating the `Acl` class:

```
use Zend\Permissions\Acl\Acl;

$acl = new Acl();
```

Once that instance is available, we can start adding roles, resources, and privileges.

For this blog post, our ACL will be used for a content-based website.

Roles

Roles are added via the `$acl->addRole()` method. This method takes either a string role name, or a `Zend\Permissions\Acl\Role\RoleInterface` instance.

Let's start with a "guest" **role**, that only allows "read" permissions.

```
use Zend\Permissions\Acl\Role\GenericRole as Role;

// Create some roles
$guest = new Role('guest');
$acl->addRole($guest);

// OR
$acl->addRole('guest');
```

> ### Referencing roles and resources
>
> Roles are simply strings. We model them as objects in zend-permissions-acl in order to provide strong typing, but the only requirement is that they return a string role name. As such, when creating permissions, you can use either a role instance, or the equivalent name.
>
> The same is true for resources, which we cover in a later section.

By default, zend-permissions-acl implements a *whitelist* approach. A *whitelist* **denies access** to everything *unless* it is explicitly *whitelisted*. (This is as opposed to a *blacklist*, where access is allowed to everything unless it is in the *blacklist*.) Unless you really know what you're doing we do not suggest toggling this; whitelists are widely regarded as a best practice for security.

What that means is that, out of the gate, while we *can* do some privilege assertions:

```
$acl->isAllowed('guest', 'blog', 'read');
$acl->isAllowed('guest', 'blog', 'write');
```

these will always return `false`, denying access. So, we need to start adding privileges.

Privileges

Privileges are assigned using `$acl->allow()`.

For the `guest` role, we'll allow the `read` privilege on *any* resource:

```
$acl->allow('guest', null, 'read');
```

The second argument to `allow()` is the resource (or resources); specifying `null` indicates the privilege applies to *all* resources. If we re-run the above assertions, we get the following:

```
$acl->isAllowed('guest', 'blog', 'read');  // true
$acl->isAllowed('guest', 'blog', 'write'); // false
```

> ## Unknown roles or resources
>
> One thing to note: if either the role or resource used with `isAllowed()` does not exist, this method *raises an exception*, specifically a `Zend\Permissions\Acl\Exception\InvalidArgumentException`, indicating the role or resource could not be found.
>
> In many situations, this may not be what you want; you may want to handle non-existent roles and/or resources gracefully. You could do this in a couple ways. First, you can test to see if the role or resource exists *before* you check the permissions, using `hasRole()` and/or `hasResource()`:
>
> ```
> if (! $acl->hasRole($foo)) {
>     // failed, due to missing role
> }
> if (! $acl->hasResource($bar)) {
>     // failed, due to missing resource
> }
> if (! $acl->isAllowed($foo, $bar, $privilege)) {
>     // failed, due to invalid privilege
> }
> ```
>
> Alternately, wrap the `isAllowed()` call in a try/catch block:
>
> ```
> try {
>     if (! $acl->isAllowed($foo, $bar, $privilege)) {
>         // failed, due to missing privileges
>     }
> } catch (AclInvalidArgumentException $e) {
>       // failed, due to missing role or resource
> }
> ```
>
> Personally, I don't like to use exceptions for application flow, so I recommend the first solution. That said, in most cases, you will be working with a role instance that you've just added to the ACL, and should only perform assertions against known resources.

Resources

Now let's add some actual resources. These are almost exactly like roles in terms of usage: you create a `ResourceInterface` instance to pass to the ACL, or, more simply, a string; resources are added via the `$acl->addResource()` method.

```php
use Zend\Permissions\Acl\Resource\GenericResource as Resource;

$resource = new Resource('blog');
$acl->addResource($resource);

// OR:
$acl->addResource('blog');
```

A resource is anything to which you want to apply permissions. In the remaining examples of this post, we'll use a "blog" as the resource, and provide a variety of permissions related to it.

Inheritance

Let's say we want to build on our previous examples, and create an "editor" role that also incorporates the permissions of the "guest" role, and adds a "write" permission to the "blog" resource.

Unlike RBAC, roles themselves contain no information about inheritance; instead, the ACL takes care of that when you add the role to the ACL:

```php
$editor = new Role('editor');
$acl->addRole($editor, $guest); // OR:
$acl->addRole($editor, 'guest');
```

The above creates a new role, `editor`, which inherits the permissions of our `guest` role. Now, let's add a privilege allowing editors to `write` to our `blog`:

```php
$acl->allow('editor', 'blog', 'write');
```

With this in place, let's do some assertions:

```php
$acl->isAllowed('editor', 'blog', 'write'); // true
$acl->isAllowed('editor', 'blog', 'read');  // true
$acl->isAllowed('guest',  'blog', 'write'); // false
```

Another role might be a "reviewer" who can "moderate" content:

```
$acl->addRole('reviewer', 'guest');
$acl->allow('reviewer', 'blog', 'moderate');

$acl->isAllowed('reviewer', 'blog', 'moderate'); // true
$acl->isAllowed('reviewer', 'blog', 'write');    // false; editor only!
$acl->isAllowed('reviewer', 'blog', 'read');     // true
$acl->isAllowed('guest',    'blog', 'moderate'); // false
```

Let's create another, an "admin" who can do **all** of the above, but also has permissions for "settings":

```
$acl->addRole('admin', ['guest', 'editor', 'reviewer']);
$acl->allow('admin', 'blog', 'settings');

$acl->isAllowed('admin',    'blog', 'settings'); // true
$acl->isAllowed('admin',    'blog', 'write');    // true
$acl->isAllowed('admin',    'blog', 'moderate'); // true
$acl->isAllowed('admin',    'blog', 'read');     // true
$acl->isAllowed('editor',   'blog', 'settings'); // false
$acl->isAllowed('reviewer', 'blog', 'settings'); // false
$acl->isAllowed('guest',    'blog', 'write');    // false
```

Note that the `addRole()` call here provides an *array* of roles as the second value this time; when called this way, the new role will inherit the privileges of *every* role listed; this allows for multiple-inheritance at the role level.

> ### Resource inheritance
>
> Resource inheritance works exactly the same as Role inheritance! Add one or more parent resources when calling `addResource()` on the ACL, and any privileges assigned to that parent resource will also apply to the new resource.
>
> As an example, I could have a "news" section in my website that has the same privilege and role schema as my blog:
>
> ```
> $acl->addResource('news', 'blog');
> ```

Fun with privileges!

Privileges are assigned using `allow()`. Interestingly, like `addRole()` and `addResource()`, the role and resource arguments presented may be *arrays* of each; in fact, so can the privileges themselves!

As an example, we could do the following:

```
$acl->allow(
    ['reviewer', 'editor'],
    ['blog', 'homepage'],
    ['write', 'maintenance']
);
```

This would assign the "write" and "maintenance" privileges on each of the "blog" and "homepage" resources to the "reviewer" and "editor" roles! Due to inheritance, the "admin" role would also gain these privileges.

Creating your ACL

When should you create your ACL, exactly? And should it contain all roles and permissions?

Typically, you will create a finite number of application or domain permissions. In our above examples, we could omit the `blog` resource and apply the ACL only within the `blog` domain (for example, only within a module of a zend-mvc or Expressive application); alternately, it could be an application-wide ACL, with resources segregated by specific domain within the application.

In either case, you will generally:

- Create a finite set of well-known roles, resources, and privileges as a global or per-domain ACL.
- Create a custom role for the current user, typically inheriting from the set of well-known roles.
- Validate the current user against the ACL.

Unlike RBAC, you typically will not add custom permissions for a user. The reason for this is due to the complexity of storing the combination of roles, resources, and privileges in a database. Storing roles is trivial:

user_id	fullname	roles
mario	Mario	editor,reviewer

You could then create the role by splitting the `roles` field and assigning each as parents:

```
$acl->addRole($user->getId(), explode(',', $user->getRoles());
```

However, for fine-grained permissions, you would essentially need an additional lookup table mapping the user to a resource and list of privileges:

user_id	resource	privileges
mario	blog	update,delete
mario	news	update

While it can be done, it is resource and code intensive.

Putting it all together, let's say the user "mario" has logged in, with the role "editor"; further, let's assume that the identity instance for our user implements `RoleInterface`. If our ACL is already populated per the above examples, I might do the following:

```
$acl->addRole($mario, $mario->getRoles());

$acl->isAllowed($mario, 'blog', 'settings'); // false; admin only!
$acl->isAllowed($mario, 'blog', 'write');    // true; all editors
$acl->isAllowed($mario, 'blog', 'read');     // true; all guests
```

Now, let's say we've gone to the work of creating the join table necessary for storing user ACL information; we might have something like the following to further populate the ACL:

```
foreach ($mario->getPrivileges() as $resource => $privileges) {
    $acl->allow($mario, $resource, explode(',', $privileges));
}
```

We could then do the following assertions:

```
$acl->isAllowed($mario,   'blog', 'update'); // true
$acl->isAllowed('editor', 'blog', 'update'); // false; mario only!
$acl->isAllowed($mario,   'blog', 'delete'); // true
$acl->isAllowed('editor', 'blog', 'delete'); // false; mario only!
```

Custom assertions

Fine-grained as the privilege system can be, sometimes it's not enough.

As an example, we may want to implement a rule that the *creator* of a content item in our website always has rights to *edit* the item. How would we implement that with the above system?

zend-permissions-acl allows you to do so via dynamic assertions. Such assertions are classes that implement `Zend\Permissions\Acl\Assertion\AssertionInterface`, which defines a single method:

```php
namespace Zend\Permissions\Assertion;

use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Resource\ResourceInterface;
use Zend\Permissions\Acl\Role\RoleInterface;

interface AssertionInterface
{
    /**
     * @return bool
     */
    public function assert(
        Acl $acl,
        RoleInterface $role = null,
        ResourceInterface $resource = null,
        $privilege = null
    );
}
```

For the sake of this example, let's assume:

- We cast our identity to a `RoleInterface` instance after retrieval.
- The content item is represented as an object.
- The object has a method `getCreatorUsername()` that will return the same username as we might have in our custom identity from the previous example.
- If the username is the same as the custom identity, allow any privileges.

Because we have PHP 7 at our disposal, we'll create the assertion as an anonymous class:

```
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Assertion\AssertionInterface;
use Zend\Permissions\Acl\Resource\ResourceInterface;
use Zend\Permissions\Acl\Role\RoleInterface;

$assertion = new class ($identity, $content) implements AssertionInterface {
    private $content;
    private $identity;

    public function __construct(RoleInterface $identity, $content)
    {
        $this->identity = $identity;
        $this->content = $content;
    }

    /**
     * @return bool
     */
    public function assert(
        Acl $acl,
        RoleInterface $role = null,
        ResourceInterface $resource = null,
        $privilege = null
    ) {
        if (null === $role || $role->getRoleId() !== $this->identity->getRoleId()) {
            return false;
        }

        if (null === $resource || 'blog' !== $resource->getResourceId()) {
            return false;
        }

        return $this->identity->getRoleId() === $this->content->getCreatorUsername();
    }
};

// Attach the assertion to all roles on the blog resource;
// custom assertions are provided as a fourth argument to allow().
$acl->allow(null, 'blog', null, $assertion);

$acl->isAllowed('mario', 'blog', 'edit'); // returns true if $mario created $content
```

The above creates a new assertion that will trigger for the "blog" resource when a privilege we do not already know about is queried. In that particular case, if the creator of our content is the same as the current user, it will return `true`, allowing access!

By creating such assertions in-place with data retrieved at runtime, you can achieve an incredible amount of flexibility for your ACLs.

Wrapping up

zend-permissions-acl provides a huge amount of power, and the ability to provide both role and resource inheritance can vastly simplify setup of complex ACLs. Additionally, the privilege system provides much-needed granularity.

If you wanted to use ACLs in middleware, the usage is quite similar to zend-permissions-rbac: inject your ACL instance in your middleware, retrieve your user identity (and thus role) from the request, and perform queries against the ACL using the current middleware or route as a resource, and either the HTTP method or the domain action you will perform as the privilege.

The main difficulty with zend-permissions-acl is that there is no 1:1 relationship between a role and a privilege, which makes storing ACL information in a database more complex. If you find yourself struggling with that fact, you may want to use RBAC instead.

Footnotes

> [1]. https://docs.zendframework.com/zend-permissions-acl/ ↩

Implement JSON-RPC with zend-json-server

by Matthew Weier O'Phinney

zend-json-server[1] provides a JSON-RPC[2] implementation. JSON-RPC is similar to XML-RPC or SOAP in that it implements a Remote Procedure Call server at a single URI using a predictable calling semantic. Like each of these other protocols, it provides the ability to introspect the server in order to determine what calls are available, what arguments each call expects, and the expected return value(s); JSON-RPC implements this via a Service Mapping Description (SMD)[3], which is usually available via an HTTP `GET` request to the server.

zend-json-server was designed to work standalone, allowing you to map a URL to a specific script that then handles the request:

```
$server = new Zend\Json\Server\Server();
$server->setClass('Calculator');

// SMD request
if ('GET' === $_SERVER['REQUEST_METHOD']) {
    // Indicate the URL endpoint, and the JSON-RPC version used:
    $server->setTarget('/json-rpc')
           ->setEnvelope(Zend\Json\Server\Smd::ENV_JSONRPC_2);

    // Grab the SMD
    $smd = $server->getServiceMap();

    // Return the SMD to the client
    header('Content-Type: application/json');
    echo $smd;
    return;
}

// Normal request
$server->handle();
```

What the above example does is:

- Create a server.
- Attach a class or object to the server. The server introspects that class in order to expose any public methods on it as calls on the server itself.
- If an HTTP `GET` request occurs, we present the service mapping description.

- Otherwise, we attempt to handle the request.

All server components in Zend Framework work similar to the above. Introspection via function or class reflection allows quickly creating and exposing services via these servers, as well as enables the servers to provide SMD, WSDL, or XML-RPC system information.

However, this approach can lead to difficulties:

- What if I need access to other application services? or want to use the fully-configured application dependency injection container?
- What if I want to be able to control the URI via a router?
- What if I want to be able to add authentication or authorization in front of the server?

In other words, ***how do I use the JSON-RPC server as part of a larger application?***

Below, I'll outline using zend-json-server in both a Zend Framework MVC application, as well as via PSR-7 middleware. In both cases, you may assume that `Acme\ServiceModel` is a class exposing public methods we wish to expose via the server.

Using zend-json-server within zend-mvc

To use zend-json-server within a zend-mvc application, you will need to:

- Provide a `Zend\Json\Server\Response` instance to the `Server` instance.
- Tell the `Server` instance to return the response.
- Populate the MVC's response from the `Server` 's response.
- Return the MVC response (which will short-circuit the view layer).

This third step requires a bit of logic, as the default response type, `Zend\Json\Server\Response\Http` , does some logic around setting headers that you'll need to duplicate.

A full example will look like the following:

```php
namespace Acme\Controller;

use Acme\ServiceModel;
use Zend\Json\Server\Response as JsonResponse;
use Zend\Json\Server\Server as JsonServer;
use Zend\Mvc\Controller\AbstractActionController;

class JsonRpcController extends AbstractActionController
{
    private $model;

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

    public function endpointAction()
    {
        $server = new JsonServer();
        $server
            ->setClass($this->model)
            ->setResponse(new JsonResponse())
            ->setReturnResponse();

        /** @var JsonResponse $jsonRpcResponse */
        $jsonRpcResponse = $server->handle();

        /** @var \Zend\Http\Response $response */
        $response = $this->getResponse();

        // Do we have an empty response?
        if (! $jsonRpcResponse->isError()
            && null === $jsonRpcResponse->getId()
        ) {
            $response->setStatusCode(204);
            return $response;
        }

        // Set the content-type
        $contentType = 'application/json-rpc';
        if (null !== ($smd = $jsonRpcResponse->getServiceMap())) {
            // SMD is being returned; use alternate content type, if present
            $contentType = $smd->getContentType() ?: $contentType;
        }

        // Set the headers and content
        $response->getHeaders()->addHeaderLine('Content-Type', $contentType);
        $response->setContent($jsonRpcResponse->toJson());
        return $response;
    }
}
```

> ### Inject your dependencies!
>
> You'll note that the above example accepts the `Acme\ServiceModel` instance via its constructor. This means that you will need to provide a factory for your controller, to ensure that it is injected with a fully configured instance — and that likely also means a factory for the model, too.
>
> To simplify this, you may want to check out the ConfigAbstractFactory[4] or ReflectionBasedAbstractFactory[5], both of which were introduced in version 3.2.0 of zend-servicemanager.

Using zend-json-server within PSR-7 middleware

Using zend-json-server within PSR-7 middleware is similar to zend-mvc:

- Provide a `Zend\Json\Server\Response` instance to the `Server` instance.
- Tell the `Server` instance to return the response.
- Create and return a PSR-7 response based on the `Server` 's response.

The code ends up looking like the following:

```php
namespace Acme\Controller;

use Acme\ServiceModel;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\EmptyResponse;
use Zend\Diactoros\Response\TextResponse;
use Zend\Json\Server\Response as JsonResponse;
use Zend\Json\Server\Server as JsonServer;

class JsonRpcMiddleware
{
    private $model;

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

    public function __invoke(
        ServerRequestInterface $request,
        ResponseInterface $response,
        callable $next
    ) {
        $server = new JsonServer();
        $server
            ->setClass($this->model)
            ->setResponse(new JsonResponse())
            ->setReturnResponse();

        /** @var JsonResponse $jsonRpcResponse */
        $jsonRpcResponse = $server->handle();

        // Do we have an empty response?
        if (! $jsonRpcResponse->isError()
            && null === $jsonRpcResponse->getId()
        ) {
            return new EmptyResponse();
        }


        // Get the content-type
        $contentType = 'application/json-rpc';
        if (null !== ($smd = $jsonRpcResponse->getServiceMap())) {
            // SMD is being returned; use alternate content type, if present
            $contentType = $smd->getContentType() ?: $contentType;
        }

        return new TextResponse(
            $jsonRpcResponse->toJson(),
            200,
            ['Content-Type' => $contentType]
        );
    }
}
```

In the above example, I use a couple of zend-diactoros[6]-specific response types to ensure that we have no extraneous information in the returned responses. I use `TextResponse` specifically, as the `toJson()` method on the zend-json-server response returns the actual JSON string, versus a data structure that can be cast to JSON.

Per the note above, you will need to configure your dependency injection container to inject the middleware instance with the model.

Summary

zend-json-server provides a flexible, robust, and simple way to create JSON-RPC services. The design of the component makes it possible to use it standalone, or within *any* application framework you might be using. Hopefully the examples above will aid you in adapting it for use within your own application!

7

Visit the zend-json-server documentation[7] to find out what else you might be able to do with this component!

Footnotes

> 1. https://docs.zendframework.com/zend-json-server/ ↩
>
> 2. http://groups.google.com/group/json-rpc/ ↩
>
> 3. http://www.jsonrpc.org/specification ↩
>
> 4. https://docs.zendframework.com/zend-servicemanager/config-abstract-factory/ ↩
>
> 5. https://docs.zendframework.com/zend-servicemanager/reflection-abstract-factory/ ↩
>
> 6. https://docs.zendframework.com/zend-diactoros ↩
>
> 7. https://docs.zendframework.com/zend-json-server/ ↩

Implement an XML-RPC server with zend-xmlrpc

by Matthew Weier O'Phinney

zend-xmlrpc[1] provides a full-featured XML-RPC[2] client and server implementation. XML-RPC is a Remote Procedure Call protocol using HTTP as the transport and XML for encoding the requests and responses.

Each XML-RPC request consists of a method call, which names the procedure (`methodName`) to call, along with its parameters. The server then returns a response, the value returned by the procedure.

As an example of a request:

```
POST /xml-rpc HTTP/1.1
Host: api.example.com
Content-Type: text/xml

<?xml version="1.0"?>
<methodCall>
    <methodName>add</methodName>
    <params>
        <param>
            <value><i4>20</i4></value>
        </param>
        <param>
            <value><i4>22</i4></value>
        </param>
    </params>
</methodCall>
```

The above is essentially requesting `add(20, 22)` from the server.

A response might look like this:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/xml

<?xml version="1.0"?>
<methodResponse>
    <params>
        <param>
            <value><i4>42</i4></value>
        </param>
    </params>
</methodResponse>
```

In the case of an error, you get a *fault* response, detailing the problem:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/xml

<?xml version="1.0"?>
<methodResponse>
    <fault>
        <value>
            <struct>
                <member>
                    <name>faultCode</name>
                    <value><int>4</int></value>
                </member>
                <member>
                    <name>faultString</name>
                    <value><string>Too few parameters.</string></value>
                </member>
            </struct>
        </value>
    </fault>
</methodResponse>
```

> ## Content-Length
>
> The specification indicates that the `Content-Length` header must be present in both requests and responses, and must be correct. I have yet to work with any XML-RPC clients or servers that followed this restriction.

Values

XML-RPC is meant to be intentionally simple, and support simple procedural operations with a limited set of allowed values. It predates JSON, but similarly defines a restricted list of allowed value types in order to allow representing almost any data structure — and note that term, *data structure*. Typed **objects** with *behavior* are never transferred, only data. (This is how SOAP differentiates from XML-RPC.)

Knowing what value types may be transmitted over XML-RPC allows you to determine whether or not it's a good fit for your web service platform.

The values allowed include:

- Integers, via either `<int>` or `<i4>` tags. (`<i4>` points to the fact that the specification restricts integers to four-byte signed integers.)
- Booleans, via `<boolean>` ; the values are either `0` or `1` .
- Strings, via `<string>` .
- Floats or doubles, via `<double>` .
- Date/Time values, in ISO-8601 format, via `<dateTime.iso8601>` .
- Base64-encoded binary values, via `<base64>` .

There are also two composite value types, `<struct>` and `<array>` . A `<struct>` contains `<member>` values, which in turn contain a `<name>` and a `<value>` :

```xml
<struct>
    <member>
        <name>minimum</name>
        <value><int>0</int></value>
    </member>
    <member>
        <name>maximum</name>
        <value><int>100</int></value>
    </member>
</struct>
```

These can be visualized as *associative arrays* in PHP.

An `<array>` consists of a `<data>` element containing any number of `<value>` items:

```xml
<array>
    <data>
        <value><int>0</int></value>
        <value><int>10</int></value>
        <value><int>20</int></value>
        <value><int>30</int></value>
        <value><int>50</int></value>
    </data>
</array>
```

The values within an array or a struct do not need to be of the same type, which makes them very suitable for translating to PHP structures.

While these values are easy enough to create and parse, doing so manually leads to a lot of overhead, particularly if you want to ensure that your server and/or client is robust. zend-xmlrpc provides all the tools to work with this

Automatically serving class methods

To simplify creating servers, zend-xmlrpc uses PHP's Reflection API[3] to scan functions and class methods in order to expose them as XML-RPC services. This allows you to add an arbitrary number of methods to your XML-RPC server, which can them be handled via a single endpoint.

In vanilla PHP, this then looks like:

```php
$server = new Zend\XmlRpc\Server;
$server->setClass('Calculator');
echo $server->handle();
```

Internally, zend-xmlrpc will take care of type conversions from the incoming request. To do so, however, you may need to document your types using slightly different notation within your docblocks. As examples, the following types do not have direct analogues in PHP:

- dateTime.iso8601
- base64
- struct

If you want to accept or return any of these types, document them:

```php
/**
 * @param dateTime.iso8601 $data
 * @param base64 $data
 * @param struct $map
 * @return base64
 */
function methodWithOddParameters($date, $data, array $map)
{
}
```

> ### Structs
>
> zend-xmlrpc *does* contain logic to determine if an array value is an indexed array or an associative array, and will generally properly convert these. However, we still recommend documenting the more specific types as noted above for purposes of using the `system.methodHelp` functionality, which is detailed below.

You may also add functions:

```
$server->addFunction('add');
```

A server can accept multiple functions and classes. However, be aware that when doing so, you need to be careful about *naming conflicts*. Fortunately, zend-xmlrpc has ways to resolve those, as well!

If you look at many XML-RPC examples, they will use method names such as `calculator.add` or `transaction.process` . zend-xmlrpc, when performing reflection, uses the method or function name by default, which will be the portion following the `.` in the previous examples. However, you can also *namespace* these, using an additional argument to either `addFunction()` or `setClass()` :

```
// Exposes Calculator methods under calculator.*:
$server->setClass('Calculator', 'calculator');

// Exposes transaction.process:
$server->addFunction('process', 'transaction');
```

This can be particularly useful when exposing multiple classes that may expose the same method names.

Server introspection

While not an official part of the standard, many servers and clients support the XML-RPC Introspection protocol[4]. The protocol defines three methods:

- `system.listMethods` , which returns a struct of methods supported by the server.
- `system.methodSignature` , which returns a struct detailing the arguments to the requested method.
- `system.methodHelp` , which returns a string description of the requested method.

The server implementation in zend-xmlrpc supports these out-of-the-box, allowing your clients to get information on exposed services!

> ### zend-xmlrpc client and introspection
>
> The client exposed within zend-xmlrpc will natively use the introspection protocol in order to provide a fluent, method-like way of invoking XML-RPC methods:
>
> ```
> $client = new Zend\XmlRpc\Client('https://xmlrpc.example.com/');
> $service = $client->getProxy();             // invokes introspection!
> $value = $service->calculator->add(20, 22); // invokes calculator.add(20, 22)
> ```

Faults and exceptions

By default, zend-xmlrpc catches exceptions in your service classes, and raises fault responses. However, these fault responses omit the exception details by default, to prevent leaking sensitive information.

You can, however, whitelist exception types with the server:

```
use App\Exception;
use Zend\XmlRpc\Server\Fault;

Fault::attachFaultException(Exception\InvalidArgumentException::class);
```

When you do so, the exception code and message will be used to generate the fault response. Note: any exception in that particular inheritance hierarchy will then be exposed as well!

Integrating with zend-mvc

The above examples all demonstrate usage in standalone scripts; what if you want to use the server inside zend-mvc?

To do so, we need to do two things differently:

- We need to create our own `Zend\XmlRpc\Request` and seed it from the MVC request content.
- We need to cast the response returned by `Zend\XmlRpc\Server::handle()` to an MVC response.

```php
namespace Acme\Controller;

use Acme\Model\Calculator;
use Zend\XmlRpc\Request as XmlRpcRequest;
use Zend\XmlRpc\Response as XmlRpcResponse;
use Zend\XmlRpc\Server as XmlRpcServer;
use Zend\Mvc\Controller\AbstractActionController;

class XmlRpcController extends AbstractActionController
{
    private $calculator;

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

    public function endpointAction()
    {
        /** @var \Zend\Http\Request $request */
        $request = $this->getRequest();

        // Seed the XML-RPC request
        $xmlRpcRequest = new XmlRpcRequest();
        $xmlRpcRequest->loadXml($request->getContent());

        // Create the server
        $server = new XmlRpcServer();
        $server->setClass($this->calculator, 'calculator');

        /** @var XmlRpcResponse $xmlRpcResponse */
        $xmlRpcResponse = $server->handle($xmlRpcRequest);

        /** @var \Zend\Http\Response $response */
        $response = $this->getResponse();

        // Set the headers and content
        $response->getHeaders()->addHeaderLine('Content-Type', 'text/xml');
        $response->setContent($xmlRpcResponse->saveXml());
        return $response;
    }
}
```

> ### Inject your dependencies!
>
> You'll note that the above example accepts the `Acme\Model\Calculator` instance via its constructor. This means that you will need to provide a factory for your controller, to ensure that it is injected with a fully configured instance — and that likely also means a factory for the model, too.
>
> To simplify this, you may want to check out the ConfigAbstractFactory[5] or ReflectionBasedAbstractFactory[6], both of which were introduced in version 3.2.0 of zend-servicemanager.

Using zend-xmlrpc's server within PSR-7 middleware

Using the zend-xmlrpc server within PSR-7 middleware is similar to zend-mvc.

```php
namespace Acme\Controller;

use Acme\Model\Calculator;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\HtmlResponse;
use Zend\XmlRpc\Request as XmlRpcRequest;
use Zend\XmlRpc\Response as XmlRpcResponse;
use Zend\XmlRpc\Server as XmlRpcServer;

class XmlRpcMiddleware
{
    private $calculator;

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

    public function __invoke(
        ServerRequestInterface $request,
        ResponseInterface $response,
        callable $next
    ) {
        // Seed the XML-RPC request
        $xmlRpcRequest = new XmlRpcRequest();
        $xmlRpcRequest->loadXml((string) $request->getBody());

        $server = new XmlRpcServer();
        $server->setClass($this->calculator, 'calculator');

        /** @var XmlRpcResponse $xmlRpcResponse */
        $xmlRpcResponse = $server->handle($xmlRpcRequest);

        return new HtmlResponse(
            $xmlRpcResponse->saveXml(),
            200,
            ['Content-Type' => 'text/xml']
        );
    }
}
```

In the above example, I use the zend-diactoros[7]-specific `HtmlResponse` type to generate the response; this could be any other response type, as long as the `Content-Type` header is set correctly, and the status code is set to 200.

Per the note above, you will need to configure your dependency injection container to inject the middleware instance with the model.

Summary

While XML-RPC may not be *du jour*, it is a tried and true method of exposing web services that has persisted for close to two decades. zend-xmlrpc's server implementation provides a flexible, robust, and simple way to create XML-RPC services around the classes and functions you define in PHP, making it possible to use it standalone, or within *any* application framework you might be using. Hopefully the examples above will aid you in adapting it for use within your own application!

Visit the zend-xmlrpc server documentation[8] to find out what else you might be able to do with this component.

Footnotes

[1]. https://docs.zendframework.com/zend-xmlrpc/ ↩

[2]. http://xmlrpc.scripting.com/spec.html ↩

[3]. http://php.net/Reflection ↩

[4]. http://xmlrpc-c.sourceforge.net/introspection.html ↩

[5]. https://docs.zendframework.com/zend-servicemanager/config-abstract-factory/ ↩

[6]. https://docs.zendframework.com/zend-servicemanager/reflection-abstract-factory/ ↩

[7]. https://docs.zendframework.com/zend-diactoros ↩

[8]. https://docs.zendframework.com/zend-xmlrpc/server/ ↩

Implement a SOAP server with zend-soap

by Matthew Weier O'Phinney

zend-soap[1] provides a full-featured SOAP[2] implementation. SOAP is an XML-based web protocol designed to allow describing *messages*, and, optionally, operations to perform. It's similar to XML-RPC, but with a few key differences:

- Arbitrary data structures may be described; you are not limited to the basic scalar, list, and struct types of XML-RPC. Messages are often serializations of specific object types on either or both the client and server. The SOAP message may include information on its own structure to allow the server or client to determine how to interpret the message.
- Multiple *operations* may be described in a message as well, versus the one call, one operation structure of XML-RPC.

In other words, it's an *extensible* protocol. This provides obvious benefits, but also a disadvantage: creating and parsing SOAP messages can quickly become quite complex!

To alleviate that complexity, Zend Framework provides the zend-soap component, which includes a server implementation.

> ## Why these articles on RPC services?
>
> We love REST; one of our projects is Apigility[3], which allows you to simply and quickly build REST APIs. However, there are occasions where RPC may be a better fit:
>
> - If your services are less *resource* oriented, and more *function* oriented (e.g., providing calculations).
> - If consumers of your services may need more uniformity in the service architecture in order to ensure they can quickly and easily consume the services, without needing to create unique tooling for each service exposed. While the goal of REST is to offer discovery, when every payload to send or receive is different, this can often lead to an explosion of code when consuming many services.
> - Some organizations and companies may standardize on certain web service protocols due to existing tooling, ability to train developers, etc.
>
> While REST may be the preferred way to architect web services, these and other reasons often dictate other approaches. As such, we provide these RPC alternatives for PHP developers.

What benefits does it offer over the PHP extension?

PHP provides SOAP client and server capabilities already via its SOAP extension[4]; why do we offer a component?

By default, PHP's `SoapServer::handle()` will:

- Grab the POST body (`php://input`), unless an XML string is passed to it.
- Emit the headers and SOAP XML response body to the output buffer.

Exceptions or PHP errors raised during processing *may* result in a SOAP fault response, with no details, or can result in invalid/empty SOAP responses returned to the client.

The primary benefit zend-soap provides, then, is *error handling*. You can whitelist exception types, and, when encountered, fault responses containing the exception details will be returned. PHP errors will be emitted as SOAP faults.

The next thing that zend-soap offers is WSDL generation. WSDL allows you to *describe* the web services you offer, so that clients know how to work with your services. ext/soap provides no functionality around creating WSDL; it simply expects that you will have a valid one for use with the client or server.

zend-soap provides an `AutoDiscover` class that uses reflection on the classes and functions you pass it in order to build a valid WSDL for you; you can then provide this to your server and your clients.

Creating a server

There are two parts to providing a SOAP server:

- Providing the server itself, which will handle requests.
- Providing the WSDL.

Building each follows the same process; you simply emit them with different HTTP `Content-Type` headers, and under different HTTP methods (the server will always react to POST requests, while WSDL should be available via GET).

First, let's define a function for populating a server instance with classes and functions:

```
use Acme\Model;

function populateServer($server, array $env)
{
    // Expose a class and its methods:
    $server->setClass(Model\Calculator::class);

    // Or expose an object instance and its methods.
    // However, this only works for Zend\Soap\Server, not AutoDiscover, so
    // should not be used here.
    // $server->setObject(new Model\Env($env));

    // Expose a function:
    $server->addFunction('Acme\Model\ping');
}
```

Note that `$server` is not type-hinted; the rationale for this decision will become more obvious soon.

Now, let's assume that the above function is available to us, and use it to create our WSDL:

```
// File /soap/wsdl.php

use Zend\Soap\AutoDiscover;

if ($_SERVER['REQUEST_METHOD'] !== 'GET') {
    // Only handle GET requests
    header('HTTP/1.1 400 Client Error');
    exit;
}

$wsdl = new AutoDiscover();
populateServer($wsdl, $_ENV);
$wsdl->handle();
```

Done! The above will emit the WSDL for either the client or server to consume.

Now, let's create the server. The server requires a few things:

- The public, HTTP-accessible location of the WSDL.
- `SoapServer` options, including the `actor` URI for the server and SOAP version targeted.

Additionally, we'll need to notify the server of its capabilities, via the `populateServer()` function.

```php
// File /soap/server.php

use Zend\Soap\Server;

if ($_SERVER['REQUEST_METHOD'] !== 'POST') {
    // Only handle POST requests
    header('HTTP/1.1 400 Client Error');
    exit;
}

$server = new Server(dirname($_SERVER['REQUEST_URI']) . '/wsdl.php', [
    'actor' => $_SERVER['REQUEST_URI'],
]);

populateServer($server, $_ENV);
$server->handle();
```

The reason for the lack of type-hint should now be clear; both the `Server` and `AutoDiscover` classes have the same API for populating the instances with classes, objects, and functions; having a common function for doing so allows us to ensure the WSDL and server do not go out of sync.

From here, you can point your clients at `/soap/server.php` on your domain, and they will have all the information they need to work with your service.

> ### setObject()
>
> `Zend\Soap\Server` also exposes a `setObject()` method, which will take an object instance, reflect it, and expose its public methods to the server. However, this method is only available in the `Server` class, not the `AutoDiscover` class.
>
> As such, if you want to create logic that can be re-used between the `Server` and `AutoDiscover` instances, you must confine your usage to `setClass()`. If that class requires constructor arguments or other ways of setting instance state, you should vary the logic for creation of the WSDL via `AutoDiscover` and creation of the server via `Server`.

Using zend-soap within a zend-mvc application

The above details an approach using vanilla PHP; what about using zend-soap within a zend-mvc context?

To do this, we'll need to learn a few more things.

First, you can provide `Server::handle()` with the *request* to process. This must be one of the following:

- a `DOMDocument`
- a `DOMNode`
- a `SimpleXMLElement`
- an object implementing `__toString()`, where that method returns an XML string
- an XML string

We can grab this information from the MVC request instance's body content.

Second, we will need the server to *return* the response, so we can use it to populate the MVC response instance. We can do that by calling `Server::setReturnResponse(true)`. When we do, `Server::handle()` will return an XML string representing the SOAP response message.

Let's put it all together:

```
namespace Acme\Controller;

use Acme\Model;
use Zend\Soap\AutoDiscover as WsdlAutoDiscover;
use Zend\Soap\Server as SoapServer;
use Zend\Mvc\Controller\AbstractActionController;

class SoapController extends AbstractActionController
{
    private $env;

    public function __construct(Model\Env $env)
    {
        $this->env = $env;
    }

    public function wsdlAction()
    {
        /** @var \Zend\Http\Request $request */
        $request = $this->getRequest();

        if (! $request->isGet()) {
            return $this->prepareClientErrorResponse('GET');
        }

        $wsdl = new WsdlAutoDiscover();
        $this->populateServer($wsdl);

        /** @var \Zend\Http\Response $response */
        $response = $this->getResponse();

        $response->getHeaders()->addHeaderLine('Content-Type', 'application/wsdl+xml')
```

```
;
        $response->setContent($wsdl->toXml());
        return $response;
    }

    public function serverAction()
    {
        /** @var \Zend\Http\Request $request */
        $request = $this->getRequest();

        if (! $request->isPost()) {
            return $this->prepareClientErrorResponse('POST');
        }

        // Create the server
        $server = new SoapServer(
            $this->url()
                ->fromRoute('soap/wsdl', [], ['force_canonical' => true]),
            [
                'actor' => $this->url()
                    ->fromRoute('soap/server', [], ['force_canonical' => true]),
            ]
        );
        $server->setReturnResponse(true);
        $this->populateServer($server);

        $soapResponse = $server->handle($request->getContent());

        /** @var \Zend\Http\Response $response */
        $response = $this->getResponse();

        // Set the headers and content
        $response->getHeaders()->addHeaderLine('Content-Type', 'application/soap+xml')
;
        $response->setContent($soapResponse);
        return $response;
    }

    private function prepareClientErrorResponse($allowed)
    {
        /** @var \Zend\Http\Response $response */
        $response = $this->getResponse();
        $response->setStatusCode(405);
        $response->getHeaders()->addHeaderLine('Allow', $allowed);
        return $response;
    }

    private function populateServer($server)
    {
        // Expose a class and its methods:
        $server->setClass(Model\Calculator::class);

        // Expose an object instance and its methods:
```

```
        $server->setObject($this->env);

        // Expose a function:
        $server->addFunction('Acme\Model\ping');
    }
}
```

The above assumes you've created routes `soap/server` and `soap/wsdl`, and uses those to generate the URIs for the server and WSDL, respectively; the `soap/server` route should map to the `SoapController::serverAction()` method and the `soap/wsdl` route should map to the `SoapController::wsdlAction()` method.

> ### Inject your dependencies!
>
> You'll note that the above example accepts the `Acme\Model\Env` instance via its constructor, allowing us to inject a fully-configured instance into the server and/or WSDL autodiscovery. This means that you will need to provide a factory for your controller, to ensure that it is injected with a fully configured instance — and that likely also means a factory for the model, too.
>
> To simplify this, you may want to check out the ConfigAbstractFactory[5] or ReflectionBasedAbstractFactory[6], both of which were introduced in version 3.2.0 of zend-servicemanager.

Using zend-soap within PSR-7 middleware

Using zend-soap in PSR-7 middleware is essentially the same as what we detail for zend-mvc: you'll need to pull the request content for the server, and use the SOAP response returned to populate a PSR-7 response instance.

The example below assumes the following:

- You are using the UrlHelper and ServerUrlHelper from zend-expressive-helpers[7] to generate URIs.
- You are routing to each middleware such that:
  - The 'soap.server' route will map to the `SoapServerMiddleware`, and only allow POST requests.
  - The 'soap.wsdl' route will map to the `WsdlMiddleware`, and only allow GET requests.

```
namespace Acme\Middleware;

use Acme\Model;
```

```php
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\TextResponse;
use Zend\Soap\AutoDiscover as WsdlAutoDiscover;
use Zend\Soap\Server as SoapServer;

trait Common
{
    private $env;

    private $urlHelper;

    private $serverUrlHelper;

    public function __construct(
        Model\Env $env,
        UrlHelper $urlHelper,
        ServerUrlHelper $serverUrlHelper
    ) {
        $this->env = $env;
        $this->urlHelper = $urlHelper;
        $this->serverUrlHelper = $serverUrlHelper;
    }

    private function populateServer($server)
    {
        // Expose a class and its methods:
        $server->setClass(Model\Calculator::class);

        // Expose an object instance and its methods:
        $server->setObject($this->env);

        // Expose a function:
        $server->addFunction('Acme\Model\ping');
    }
}

class SoapServerMiddleware
{
    use Common;

    public function __invoke(
        ServerRequestInterface $request,
        ResponseInterface $response,
        callable $next
    ) {
        $server = new SoapServer($this->generateUri('soap.wsdl'), [
            'actor' => $this->generateUri('soap.server')
        ]);
        $server->setReturnResponse(true);
        $this->populateServer($server);

        $xml = $server->handle((string) $request->getBody());
        return new TextResponse($xml, 200, [
            'Content-Type' => 'application/soap+xml',
        ]);
    }

    private function generateUri($route)
    {
        return ($this->serverUrlHelper)(
            ($this->urlHelper)($route)
        );
    }
}

class WsdlMiddleware
{
    use Common;

    public function __invoke(
        ServerRequestInterface $request,
        ResponseInterface $response,
        callable $next
    ) {
        $server = new WsdlAutoDiscover();
        $this->populateServer($server);

        return new TextResponse($server->toXml(), 200, [
            'Content-Type' => 'application/wsdl+xml',
        ]);
    }
}
```

Since each middleware has the same basic construction, I've created a trait with the common functionality, and composed it into each middleware. As you will note, the actual work of each middleware is relatively simple; create a server, and marshal a resposne to return.

In the above example, I use the zend-diactoros[8]-specific `TextResponse` type to generate the response; this could be any other response type, as long as the `Content-Type` header is set correctly, and the status code is set to 200.

Per the note above, you will need to configure your dependency injection container to inject the middleware instances with the model and helpers.

Summary

While SOAP is often maligned in PHP circles, it is still in wide use within enterprises, and used in many cases to provide cross-platform web services with predictable behaviors. It can be quite complex, but zend-soap helps smooth out the bulk of the complexity. You can use it standalone, within a Zend Framework MVC application, or within *any* application framework you might be using.

Visit the zend-soap documentation[9] to find out what else you might be able to do with this component.

Footnotes

> 1. https://docs.zendframework.com/zend-soap/ ↩

> 2. https://en.wikipedia.org/wiki/SOAP ↩

> 3. https://apigility.org ↩

> 4. http://php.net/soap ↩

> 5. https://docs.zendframework.com/zend-servicemanager/config-abstract-factory/ ↩

> 6. https://docs.zendframework.com/zend-servicemanager/reflection-abstract-factory/ ↩

> 7. https://docs.zendframework.com/zend-expressive/features/helpers/url-helper ↩

> 8. https://docs.zendframework.com/zend-diactoros ↩

> 9. https://docs.zendframework.com/zend-soap/ ↩

Context-specific escaping with zend-escaper

by Matthew Weier O'Phinney

Security of your website is not just about mitigating and preventing things like SQL injection; it's also about protecting your users as they browse the site from things like cross-site scripting (XSS) attacks, cross-site request forgery (CSRF), and more. In particular, you need to be very careful about how you generate HTML, CSS, and JavaScript to ensure that you do not create such vectors.

As the mantra goes, filter input, and *escape output*.

Believe it or not, escaping in PHP is not terribly easy to get right. For example, to properly escape HTML, you need to use `htmlspecialchars()`, with the flags `ENT_QUOTES | ENT_SUBSTITUTE`, and provide a character encoding. Who really wants to write

```
htmlspecialchars($string, ENT_QUOTES | ENT_SUBSTITUTE, 'utf-8')
```

every single time they need to escape a string for use in HTML?

Escaping HTML attributes, CSS, and JavaScript each require a regular expression to identify known problem strings, and a number of heuristics to replace unicode characters with hex entities, each with different rules. While much of this can be done with built-in PHP features, these features do not catch all potential attack vectors. A comprehensive solution is required.

Zend Framework provides the zend-escaper[1] component to manage this complexity for you, exposing functionality for escaping HTML, HTML attributes, JavaScript, CSS, and URLs to ensure they are safe for the browser.

Installation

zend-escaper only requires PHP (of at least version 5.5 at the time of writing), and is installable via composer:

```
$ composer require zendframework/zend-escaper
```

Usage

While we considered making zend-escaper act as either functions or static methods, there was one thing in the way: proper escaping requires knowledge of the intended output character set. As such, `Zend\Escaper\Escaper` must first be instantiated with the output character set; once it has, you call methods on it.

```php
use Zend\Escaper\Escaper;

$escaper = new Escaper('iso-8859-1');
```

By default, if no character set is provided, it assumes `utf-8`; we recommend using UTF-8 unless there is a compelling reason not to. As such, in most cases, you can instantiate it with no arguments:

```php
use Zend\Escaper\Escaper;

$escaper = new Escaper();
```

The class provides five methods:

- `escapeHtml(string $html) : string` will escape the string so it may be safely used as HTML. In general, this means `<`, `>`, and `&` characters (as well as others) are escaped to prevent injection of unwanted tags and entities.
- `escapeHtmlAttr(string $value) : string` escapes a string so it may safely be used within an HTML attribute value.
- `escapeJs(string $js) : string` escapes a string so it may safely be used within a `<script>` tag. In particular, this ensures that the code injected cannot contain continuations and escape sequences that lead to XSS vectors.
- `escapeCss(string $css) : string` escapes a string to use as CSS within `<style>` tags; similar to JS, it prevents continuations and escape sequences that can lead to XSS vectors.
- `escapeUrl(string $urlPart) : string` escapes a string to use *within* a URL; it should not be used to escape the entire URL itself. It *should* be used to escape things such as the URL path, query string parameters, and fragment, however.

So, as examples:

```
echo $escaper->escapeHtml('<script>alert("zf")</script>');
// results in "&lt;script&gt;alert(&quot;zf&quot;)&lt;/script&gt;"

echo $escaper->escapeHtmlAttr("<script>alert('zf')</script>");
// results in "&lt;script&gt;alert&#x28;&#x27;zf&#x27;&#x29;&lt;&#x2F;script&gt;"

echo $escaper->escapeJs("bar&quot;; alert(&quot;zf&quot;); var xss=&quot;true");
// results in "bar\x26quot\x3B\x3B\x20alert\x28\x26quot\x3Bzf\x26quot\x3B\x29\x3B\x20v
ar\x20xss\x3D\x26quot\x3Btrue"

echo $escaper->escapeCss("background-image: url('/zf.png?</style><script>alert(\'zf\')
</script>');");
// results in "background\2D image\3A \20 url\28 \27 \2F zf\2E png\3F \3C \2F style\3E
 \3C script\3E alert\28 \5C \27 zf\5C \27 \29 \3C \2F script\3E \27 \29 \3B"

echo $escaper->escapeUrl('/foo " onmouseover="alert(\'zf\')');
// results in "%2Ffoo%20%22%20onmouseover%3D%22alert%28%27zf%27%29"
```

As you can see from these examples, the component aggresively filters each string to ensure it is escaped correctly for the context for which it is intended.

How and where might you use this?

- Within templates, to ensure output is properly escaped. For example, zend-view[2] includes helpers for it; it would be easy to add such functionality to Plates[3] and other templating solutions.
- In email templates.
- In serializers for APIs, to ensure things like URLs or XML attribute data are properly escaped.
- In error handlers, to ensure error messages are escaped and do not contain XSS vectors.

The main point is that escaping *can* be easy with zend-escaper; start securing your output today!

Footnotes

> 1. https://docs.zendframework.com/zend-escaper ↩

> 2. https://docs.zendframework.com/zend-view ↩

> 3. http://platesphp.com ↩

Filter input using zend-filter

by Matthew Weier O'Phinney

When securing your website, the mantra is "Filter input, escape output." We covered escaping output in the Context-specific escaping with zend-escaper article. We're now going to turn to *filtering input*.

Filtering input is rather complex, and spans a number of practices:

- Filtering/normalizing input. As an example, your web page may have a form that allows submitting a credit card number. These have a variety of formats that may include spaces or dashes or dots — but the only characters that are of importance are the *digits*. As such, you will want to *normalize* such input to strip out the unwanted characters.
- Validating input. Once you have done such normalization, you can then check to see that the data is actually *valid* for its context. This may include one or more rules. Using our credit card example, you might first check it is of an appropriate length, and then verify that it begins with a known vendor digit, and only after those pass, validate the number against a online service.

For now, we're going to look at the first item, filtering and normalizing input, using the component zend-filter[1].

Installation

To install zend-filter, use Composer:

```
$ composer require zendframework/zend-filter
```

Currently, the only *required* dependency is zend-stdlib. However, a few other components are suggested, based on which filters and/or featurse you may want to use:

- zendframework/zend-servicemanager is used by the `FilterChain` component for looking up filters by their *short name* (versus fully qualified class name).
- zendframework/zend-crypt is used by the encryption and decryption filters.
- zendframework/zend-uri is used by the `UriNormalize` filter.
- zendframework/zend-i18n is used by several filters that provide internationalization features.

For our examples, we'll be using the `FilterChain` functionality, so we will also want to install zend-servicemanager:

```
$ composer require zendframework/zend-servicemanager
```

FilterInterface

Filters can be one of two things: a callable that accepts a single argument (the value to filter), or an instance of `Zend\Filter\FilterInterface`:

```
namespace Zend\Filter;

interface FilterInterface
{
    public function filter($value);
}
```

The value can be literally anything, and the filter can return anything itself. Generally speaking, if a filter cannot operate on the value, it is expected to return it verbatim.

zend-filter provides a few dozen filters for common operations, including things like:

- Normalizing strings, integers, etc. to their corresponding boolean values.
- Normalizing strings representing integers to integer values.
- Normalizing empty values to null values.
- Normalizing input sets representing date and/or time selections from forms to `DateTime` instances.
- Normalizing URI values.
- Comparing values to whitelists and blacklists.
- Trimming whitespace, stripping newlines, and removing HTML tags or entities.
- Upper and lower casing words.
- Stripping everything but digits.
- Performing PCRE regexp replacements.
- Word inflection (camel-case to underscores and vice versa, etc.).
- Decrypting and encrypting file contents, as well as casting file contents to lower or upper case.
- Compressing and decompressing values.
- Decrypting and encrypting values.

Any of these may be used by themselves. However, in most cases, if that's all you're doing, you might as well just do the functionality inline. So, what's the benefit of zend-filter?

Chaining filters!

FilterChain

When we get input from the web, it generally comes as strings, and is the result of user input. As such, we often get a lot of garbage: extra spaces, unnecessary newlines, HTML characters, etc.

When filtering such input, we might want to perform several operations:

```
$value = $request->getParsedBody()['phone'] ?? '';
$value = trim($value);
$value = preg_replace("/[^\n\r]/", '', $value);
$value = preg_replace('/[^\d]/', '', $value);
```

We then need to test our code to ensure that we're filtering correctly. Additionally, if at any point we fail to re-assign, we may lose the changes we were performing!

With zend-filter, we can instead use a `FilterChain`. The above example becomes:

```
use Zend\Filter\FilterChain;

$filter = new FilterChain();
// attachByName uses the class name, minus the namespace, and
$filter->attachByName('StringTrim');
$filter->attachByName('StripNewlines');
$filter->attachByName('Digits');
$value = $filter->filter($request->getParsedBody()['phone'] ?? '');
```

Here's another example: let's say we have configuration keys that are in `snake_case_format`, and which may be read from a file, and we wish to convert those values to `CamelCase`.

```
use Zend\Filter;

$filter = new Filter\FilterChain();
// attach lets you provide the instance you wish to use; this will work
// even without zend-servicemanager installed.
$filter->attach(new Filter\StringTrim());
$filter->attach(new Filter\StripNewlines()); // because we may have \r characters
$filter->attach(new Filter\Word\UnderscoreToCamelCase());

$configKeys = array_map([$filter, 'filter'], explode("\n", $fileContents));
```

This new example demonstrates a key feature of a `FilterChain`: you can re-use it! Instead of having to put the code for normalizing the values within an `array_map` callback, we can instead directly use our already configured `FilterChain`, invoking it once for each value!

Wrapping up

zend-filter can be a powerful tool in your arsenal for dealing with user input. Paired with good validation, you can protect your application from malicious or malformed input.

Footnotes

> 1. https://docs.zendframework.com/zend-filter/ ↩

Validate input using zend-validator

by Matthew Weier O'Phinney

In our article Filter input using zend-filter, we covered filtering data. The filters in zend-filter are generally used to *pre-filter* or *normalize* incoming data. This is all well and good, but we still don't know if the data is *valid*. That's where zend-validator comes in.

Installation

To install zend-validator, use Composer:

```
$ composer require zendframework/zend-validator
```

Like zend-filter, the only *required* dependency is zend-stdlib. However, a few other components are suggested, based on which filters and/or features you may want to use:

- zendframework/zend-servicemanager is used by the `ValidatorPluginManager` and `ValidatorChain` to look up validators by their *short name* (versus fully qualified class name), as well as to allow usage of validators with dependencies.
- zendframework/zend-db is used by a pair of validators that can check if a matching record exists (or does not!).
- zendframework/zend-uri is used by the `Uri` validator.
- The CSRF validator requires both zendframework/zend-math and zendframework/zend-session.
- zendframework/zend-i18n and zendframework/zend-i18n-resources can be installed in order to provide translation of validation error messages.

For our examples, we'll be using the `ValidatorChain` functionality with a `ValidatorPluginManager`, so we will also want to install zend-servicemanager:

```
$ composer require zendframework/zend-servicemanager
```

ValidatorInterface

The current incarnation of zend-validator is *stateful*; validation error messages are stored in the validator itself. As such, validators must implement the `ValidatorInterface`:

```
namespace Zend\Validator;

interface ValidatorInterface
{
    /**
     * @param mixed $value
     * @return bool
     */
    public function isValid($value);

    /**
     * @return array
     */
    public function getMessages();
}
```

The `$value` can be literally anything; a validator examines it to see if it is valid, and returns a boolean result. If it is invalid, a subsequent call to `getMessages()` should return an associative array with the keys being message identifiers, and the values the human-readable message strings.

As such, usage looks like the following:

```
if (! $validator->isValid($value)) {
    // Invalid value
    echo "Failed validation:\n";
    foreach ($validator->getMessages() as $message) {
        printf("- %s\n", $message);
    }
    return false;
}
// Valid value!
return true;
```

Stateless validations are planned

At the time of writing, we have proposed[1] a new validation component to work in parallel with zend-validator; this new component will implement a *stateless* architecture. Its proposed validation interface will no longer return a boolean, but rather a `ValidationResult`. That instance will provide a method for determining if the validation was successful, encapsulate the value that was validated, and, for invalid values, provide access to the validation error messages. Doing so will allow better re-use of validators within the same execution process.

This proposal also includes code for adapting existing zend-validator implementations to work with the stateless design.

zend-validator provides a few dozen filters for common operations, including things like:

- Common conditionals like `LessThan`, `GreaterThan`, `Identical`, `NotEmpty`, `IsInstanceOf`, `InArray`, and `Between`.
- String values, such as `StringLength`, `Regex`.
- Network-related values such as `Hostname`, `Ip`, `Uri`, and `EmailAddress`.
- Business values such as `Barcode`, `CreditCard`, `GpsPoint`, `Iban`, and `Uuid`.
- Date and time related values such as `Date`, `DateStep`, and `Timezone`.

Any of these validators may be used by themselves.

In many cases, though, your validation may be related to a *set* of validations: as an example, the value must be non-empty, a certain number of characters, and fulfill a regular expression. Like filters, zend-validator allows you to do this with *chains*.

ValidatorChain

Usage of a validator chain is similar to filter chains: attach validators you want to execute, and then pass the value to the chain:

```
use Zend\Validator;

$validator = new Validator\ValidatorChain();
$validator->attach(new Validator\NotEmpty());
$validator->attach(new Validator\StringLength(['min' => 6]));
$validator->attach(new Validator\Regex('/^[a-f0-9]{6,12}$/');

if (! $validator->isValid($value)) {
    // Failed validation
    var_dump($validator->getMessages());
}
```

The above uses validator instances, eliminating the need for `ValidatorPluginManager`, and thus avoids usage of zend-servicemanager. However, if we have zend-servicemanager installed, we can replace usage of `attach()` with `attachByName()`:

```
use Zend\Validator;

$validator = new Validator\ValidatorChain();
$validator->attachByName('NotEmpty');
$validator->attachByName('StringLength', ['min' => 6]);
$validator->attachByName('Regex', ['pattern' => '/^[a-f0-9]{6,12}$/']);

if (! $validator->isValid($value)) {
    // Failed validation
    var_dump($validator->getMessages());
}
```

Breaking the chain

If you were to run either of these examples with `$value = ''`, you may discover something unexpected: you'll get validation error messages *for every single validator*! This seems wasteful; there's no need to run the `StringLength` or `Regex` validators if the value is empty, is there?

To solve this problem, when attaching a validator, we can tell the chain to *break execution* if the given validator fails. This is done by passing a boolean flag:

- as the second argument to `attach()`
- as the third argument to `attachByName()` (the second argument is an array of constructor options)

Let's update the second example:

```
use Zend\Validator;

$validator = new Validator\ValidatorChain();
$validator->attachByName('NotEmpty', [], $breakChainOnFailure = true);
$validator->attachByName('StringLength', ['min' => 6], true);
$validator->attachByName('Regex', ['pattern' => '/^[a-f0-9]{6,12}$/']);

if (! $validator->isValid($value)) {
    // Failed validation
    var_dump($validator->getMessages());
}
```

The above adds a boolean `true` as the `$breakChainOnFailure` argument to the `attachByName()` method calls of the `NotEmpty` and `StringLength` validators (we had to provide an empty array of options for the `NotEmpty` validator so we could pass the flag). In these cases, if the value fails validation, no further validators will be executed.

Thus:

- `$value = ''` will result in a single validation failure message, produced by the `NotEmpty` validator.
- `$value = 'test'` will result in a single validation failure message, produced by the `StringLength` validator.
- `$value = 'testthis'` will result in a single validation failure message, produced by the `Regex` validator.

Prioritization

Validators are executed in the same order in which they are attached to the chain by default. However, internally, they are stored in a `PriorityQueue`; this allows you to provide a specific order in which to execute the validators. Higher values execute earlier, while lower values (including negative values) execute last. The default priority is 1.

Priority values may be passed as the third argument to `attach()` and fourth argument to `attachByName()`.

As an example:

```
$validator = new Validator\ValidatorChain();
$validator->attachByName('StringLength', ['min' => 6], true, 1);
$validator->attachByName('Regex', ['pattern' => '/^[a-f0-9]{6,12}$/'], false, -100);
$validator->attachByName('NotEmpty', [], true, 100);
```

In the above, when executing the validation chain, the order will still be `NotEmpty`, followed by `StringLength`, followed by `Regex`.

> ### Why prioritize?
>
> Why would you use this feature? The main reason is if you want to define validation chains via configuration, and cannot guarantee the order in which the items will be present in configuration. By adding a priority value, you can ensure that recreation of the validation chain will preserve the expected order.

Context

Sometimes we may want to vary how we validate a value based on whether or not another piece of data is present, or based on that other piece of data's value. zend-validator offers an unofficial API for that, via an optional `$context` value you can pass to `isValid()`. The

`ValidatorChain` accepts this value, and, if present, will pass it to each validator it composes.

As an example, let's say you want to capture an email address (form field "contact"), but only if the user has selected a radio button allowing you to do so (form field "allow_contact"). We might write that validator as follows:

```
use ArrayAccess;
use ArrayObject;
use Zend\Validator\EmailAddress;
use Zend\Validator\ValidatorInterface;

class ContactEmailValidator implements ValidatorInterface
{
    const ERROR_INVALID_EMAIL = 'contact-email-invalid';

    /** @var string */
    private $contextVariable;

    /** @var EmailAddress */
    private $emailValidator;

    /** @var string[] */
    private $messages = [];

    /** @var string[] */
    private $messageTemplates = [
        self::ERROR_INVALID_EMAIL => 'Email address "%s" is invalid',
    ];

    public function __construct(
        EmailAddress $emailValidator = null,
        string $contextVariable = 'allow_contact'
    ) {
        $this->emailValidator = $emailValidator ?: new EmailAddress();
        $this->contextVariable = $contextVariable;
    }

    public function isValid($value, $context = null)
    {
        $this->messages = [];

        if (! $this->allowsContact($context)) {
            // Value will be discarded, so always valid.
            return true;
        }

        if ($this->emailValidator->isValid($value)) {
            return true;
        }

        $this->messages[self::ERROR_INVALID_EMAIL] = sprintf(
            $this->messageTemplates[self::ERROR_INVALID_EMAIL],
            var_export($value, true)
        );
        return false;
    }

    public function getMessages()
    {
        return $this->messages;
    }

    private function allowsContact($context) : bool
    {
        if (! $context ||
            ! (is_array($context)
              || $context instanceof ArrayObject
              || $context instanceof ArrayAccess)
        ) {
            return false;
        }

        $allowsContact = $context[$this->contextVariable] ?? false;

        return (bool) $allowsContact;
    }
}
```

We would then add it to the validator chain, and call it like so:

```
$validator->attach(new ContactEmailValidator());
if (! $validator->isValid($data['contact'], $data)) {
    // Failed validation!
}
```

This approach can allow for some quite complex validation routines, particularly if you nest validation chains within custom validators!

Registering your own validators.

If you write your own validators, chances are you'll want to use them with the `ValidatorChain`. This class composes a `ValidatorPluginManager`, which is a plugin manager built on top of zend-servicemanager. As such, you can register your validators with it:

```
$plugins = $validator->getPluginManager();
$plugins->setFactory(ContactEmailValidator::class, ContactEmailValidatorFactory::class
);
$plugins->setService(ContactEmailValidator::class, $contactEmailValidator);
```

Alternately, if using zend-mvc or Expressive, you can provide configuration via the `validators` configuration key:

```
return [
    'validators' => [
        'factories' => [
            ContactEmailValidator::class => ContactEmailValidatorFactory::class,
        ],
    ],
];
```

If you want to use a "short name" to identify your validator, we recommend using an alias, aliasing the short name to the fully qualified class name.

Wrapping up

Between using zend-filter to normalize and pre-filter values, and zend-validator to validate the values, you can start locking down the input your users submit to your application.

That said, what we've demonstrated so far is how to work with single values. Most forms submit *sets* of values; using the approaches so far can lead to a lot of code!

We have a solution for this as well, via our zend-inputfilter component. Read the article Validate data using zend-inputfilter for more information.

Footnotes

1. https://discourse.zendframework.com/t/rfc-new-validation-component/208/ ↩

Validate data using zend-inputfilter

by Matthew Weier O'Phinney

In our articles Filter input using zend-filter and Validate input using zend-validator, we covered the usage of zend-filter and zend-validator. With these two components, you now have the tools necessary to ensure any given user input is valid, fulfilling the first half of the "filter input, escape output" mantra.

However, as we discussed in the zend-validator article, as powerful as validation chains are, they only allow you to validate a *single* value at a time. How do you go about validating *sets* of values — such as data submitted from a form, or a resource for an API?

To solve that problem, Zend Framework provides zend-inputfilter[1]. An *input filter* aggregates one or more *inputs*, any one of which may also be another input filter, allowing you to validate complex, multi-set, and nested set values.

Installation

To install zend-inputfilter, use Composer:

```
$ composer require zendframework/zend-inputfilter
```

zend-inputfilter only directly requires zend-filter, zend-stdlib, and zend-validator. To use its powerful factory feature, however, you'll also need zend-servicemanager, as it greatly simplifies creation of input filters:

```
$ composer require zendframework/zend-servicemanager
```

Theory of operation

An input filter composes one or more inputs, any of which may also be an input filter (and thus represent a *set* of data values).

Any given input is considered *required* by default, but can be configured to be *optional*. When required, an input will be considered invalid if the value is not present in the data set, or is empty. When optional, if the value is not present, or is empty, it is considered valid. An additional flag, `allow_empty`, can be used to allow empty values for required elements; still another flag, `continue_if_empty`, will force validation to occur for either required or optional values if the value is present but empty.

When validating a value, two steps occur:

- The value is passed to a *filter chain* in order to normalize the value. Typical normalizations include stripping non-digit characters for phone numbers and credit card numbers; trimming whitespace; etc.
- The value is then passed to a *validator chain* to determine if the normalized value is valid.

An input filter aggregates the inputs, as well as the *values* themselves. You pass the user input to the input filter after it has been configured, and then check to see if it is valid. If it is, you can pull the *normalized* values from it (as well as the raw values, if desired). If any value is invalid, you would then pull the validation error messages from it.

> ### Stateless operation
>
> The current approach is *stateful*: values are passed to the input filter before you execute its `isValid()` method, and then the values and any validation error messages are stored within the input filter instance for later retrieval. This can cause issues if you wish to use the same input filter multiple times in the same request.
>
> For this reason, we are planning a new, parallel component that provides *stateless* validation: calling `isValid()` will require passing the value(s) to validate, and both inputs and input filters alike will return a result object from this method with the raw and normalized values, the result of validation, and any validation error messages.

Getting started

Let's consider a registration form where we want to capture a user email and their password. In our first example, we will use explicit usage, which does not require the use of plugin managers.

```php
use Zend\Filter;
use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;
use Zend\Validator;

$email = new Input('email');
$email->getFilterChain()
        ->attach(new Filter\StringTrim());
$email->getValidatorChain()
        ->attach(new Validator\EmailAddress());

$password = new Input('password');
$password->getValidatorChain()
        ->attach(new Validator\StringLength(8), true)
      ->attach(new Validator\Regex('/[a-z]/'))
      ->attach(new Validator\Regex('/[A-Z]/'))
      ->attach(new Validator\Regex('/[0-9]/'))
      ->attach(new Validator\Regex('/[.!@#$%^&*;:]/'));

$inputFilter = new InputFilter();
$inputFilter->add($email);
$inputFilter->add($password);
$inputFilter->setData($_POST);

if ($inputFilter->isValid()) {
    echo "The form is valid\n";
    $values = $inputFilter->getValues();
} else {
    echo "The form is not valid\n";
    foreach ($inputFilter->getInvalidInput() as $error) {
        var_dump($error->getMessages());
    }
}
```

The above creates two inputs, one each for the incoming email address and password. The email address will be *trimmed* of whitespace, and then validated. The password will be validated only, checking that we have a value of at least 8 characters, with at least one each of lowercase, uppercase, digit, and special characters. Further, if any given character is missing, we'll get a validation error message so that the user knows how to create their password.

Each input is added to an input filter instance. We pass the form data (via the `$_POST` superglobal), and then check to see if it is valid. If so, we grab the values from it (we can get the original values via `getRawValues()`). If not, we grab error messages from it.

By default, all inputs are considered *required*. Let's say we also wanted to collect the user's full name, but make that optional. We could create an input like the following:

```
$name = new Input('user_name');
$name->setRequired(false); // OPTIONAL!
$name>getFilterChain()
        ->attach(new Filter\StringTrim());
```

Input specifications

As noted in the "Installation" section, we can leverage zend-servicemanager and the various plugin managers composed in it in order to create our filters.

`Zend\InputFilter\InputFilter` internally composes `Zend\InputFilter\Factory`, which itself composes:

- `Zend\InputFilter\InputFilterPluginManager`, a plugin manager for managing `Zend\InputFilter\Input` and `Zend\InputFilter\InputFilter` instances.
- `Zend\Filter\FilterPluginManager`, a plugin manager for filters.
- `Zend\Validator\ValidatorPluginManager`, a plugin manager for validators.

The upshot is that we can often use *specifications* instead of *instances* to create our inputs and input filters.

As such, our above examples can be written like this:

```php
use Zend\InputFilter\InputFilter;

$inputFilter = new InputFilter();
$inputFilter->add([
    'name' => 'email',
    'filters' => [
        ['name' => 'StringTrim']
    ],
    'validators' => [
        ['name' => 'EmailAddress']
    ],
]);
$inputFilter->add([
    'name' => 'user_name',
    'required' => false,
    'filters' => [
        ['name' => 'StringTrim']
    ],
]);
$inputFilter->add([
    'name' => 'password',
    'validators' => [
        [
            'name' => 'StringLength',
            'options' => ['min' => 8],
            'break_chain_on_failure' => true,
        ],
        ['name' => 'Regex', 'options' => ['pattern' => '/[a-z]/'],
        ['name' => 'Regex', 'options' => ['pattern' => '/[A-Z]/'],
        ['name' => 'Regex', 'options' => ['pattern' => '/[0-9]/'],
        ['name' => 'Regex', 'options' => ['pattern' => '/[.!@#$%^&*;:]/'],
    ],
]);
```

There are a number of other fields you could use:

- `type` allows you to specify the input or input filter class to use when creating the input.
- `error_message` allows you to specify a single error message to return for an input on validation failure. This is often useful as otherwise you'll get an array of messages for each input.
- `allow_empty` and `continue_if_empty`, which were discussed earlier, and control how validation occurs when empty values are encountered.

Why would you do this instead of using the programmatic interface, though?

First, this approach leverages the various plugin managers, which means that any given input, input filter, filter, or validator will be pulled from their respective plugin manager. This allows you to provide additional types easily, but, more importantly, override existing types.

Second, the configuration-based approach allows you to store the definitions in configuration, and potentially even *override* the definitions via configuration merging! Apigility utilizes this feature heavily, in part to provide different input filters based on API version.

Managing the plugin managers

To ensure that you can use already configured plugin managers, you can inject them into the `Zend\InputFilter\Factory` composed in your input filter. As an example, considering the following service factory for an input filter:

```
function (ContainerInterface $container)
{
    $filters = $container->get('FilterManager');
    $validators = $container->get('ValidatorManager');
    $inputFilters = $container->get('InputFilterManager');

    $inputFilter = new InputFilter();
    $inputFilterFactory = $inputFilter->getFactory();
    $inputFilterFactory->setDefaultFilterChain($filters);
    $inputFilterFactory->setDefaultValidatorChain($validators);
    $inputFilterFactory->setInputFilterManager($inputFilters);

    // add inputs to the $inputFilter, and finally return it...
    return $inputFilter;
}
```

Managing Input Filters

The `InputFilterPluginManager` allows you to define input filters with dependencies, which gives you the ability to create re-usable, complex input filters. One key aspect to using this feature is that the `InputFilterPluginManager` also ensures the configured filter and validator plugin managers are injected in the factory used by the input filter, ensuring any overrides or custom filters and validators you've defined are present.

To make this work, the base `InputFilter` implementation also implements `Zend\Stdlib\InitializableInterface`, which defines an `init()` method; the `InputFilterPluginManager` calls this *after* instantiating your input filter and injecting it with a factory composing all the various plugin manager services.

What this means is that if you use this method to `add()` your inputs and nested input filters, everything will be properly configured!

As an example, let's say we have a "transaction_id" field, and that we need to check if that transaction identifier exists in the database. As such, we may have a custom validator that depends on a database connection to do this. We could write our input filter as follows:

```
namespace MyBusiness;

use Zend\InputFilter\InputFilter;

class OrderInputFilter extends InputFilter
{
    public function init()
    {
        $this->add([
            'name' => 'transaction_id',
            'validators' => [
                ['name' => TransactionIdValidator::class],
            ],
        ]);
    }
}
```

We would then register this in our `input_filters` configuration:

```
// in config/autoload/input_filters.global.php
return [
    'input_filters' => [
        'invokables' => [
            MyBusiness\OrderInputFilter::class => MyBusiness\OrderInputFilter::class,
        ],
    ],
    'validators' => [
        'factories' => [
            MyBusiness\TransactionIdValidator::class => MyBusiness\TransactionIdValida
torFactory::class,
        ],
    ],
];
```

This approach works best with the *specification* form; otherwise you need to pull the various plugin managers from the composed factory and pass them to the individual inputs:

```
$transId = new Input();
$transId->getValidatorChain()
    ->setValidatorManager($this->getFactory()->getValidatorManager());
$transId->getValidatorChain()
    ->attach(TransactionIdValidator::class);
```

Specification-driven input filters

Finally, we can look at *specification-driven* input filters.

The component provides an `InputFilterAbstractServiceFactory`. When you request an input filter or input that is not directly in the `InputFilterPluginManager`, this abstract factory will then check to see if a corresponding value is present in the `input_filter_specs` configuration array. If so, it will pass that specification to a `Zend\InputFilter\Factory` configured with the various plugin managers in order to create the instance.

Using our original example, we could define the registration form input filter as follows:

```
return [
    'input_filter_specs' => [
        'registration_form' => [
            [
                'name' => 'email',
                'filters' => [
                    ['name' => 'StringTrim']
                ],
                'validators' => [
                    ['name' => 'EmailAddress']
                ],
            ],
            [
                'name' => 'user_name',
                'required' => false,
                'filters' => [
                    ['name' => 'StringTrim']
                ],
            ],
            [
                'name' => 'password',
                'validators' => [
                    [
                        'name' => 'StringLength',
                        'options' => ['min' => 8],
                        'break_chain_on_failure' => true,
                    ],
                    ['name' => 'Regex', 'options' => ['pattern' => '/[a-z]/'],
                    ['name' => 'Regex', 'options' => ['pattern' => '/[A-Z]/'],
                    ['name' => 'Regex', 'options' => ['pattern' => '/[0-9]/'],
                    ['name' => 'Regex', 'options' => ['pattern' => '/[.!@#$%^&*;:]/'],
                ],
            ],
        ],
    ],
];
```

We would then retrieve it from the input filter plugin manager:

```
$inputFilter = $inputFilters->get('registration_form');
```

Considering most input filters do not need to compose dependencies other than the inputs and input filters they aggregate, this approach makes for a dynamic way to define input validation.

Topics not covered

zend-inputfilter has a ton of other features as well:

- Input and input filter *merging*.
- Handling of array values.
- Collections (repeated data sets of the same structure).
- Filtering of file uploads.

On top of all this, it provides a number of interfaces against which you can program in order to write completely custom functionality!

One huge strength of zend-inputfilter is that it can be used for any sort of data set you need to validate: forms, obviously, but also API payloads, data retrieved from a message queue, and more.

Footnotes

[1]. https://docs.zendframework.com/zend-inputfilter ↩

End-to-end encryption with Zend Framework 3

by Enrico Zimuel

zend-crypt[1] 3.1.0 includes a hybrid cryptosystem[2], a feature that can be used to implement an end-to-end encryption[3] schema in PHP.

A hybrid cryptosystem is a cryptographic mechanism that uses symmetric encyption (e.g. AES[4]) to encrypt a message, and public-key cryptography (e.g. RSA[5]) to protect the encryption key. This methodology guarantee two advantages: the speed of a symmetric algorithm and the security of public-key cryptography.

Before I present the PHP implementation, let's explore the hybrid mechanism in more detail. Below is a diagram demonstrating a hybrid encryption schema:

A user (the *sender*) wants to send a protected message to another user (the *receiver*). He/she generates a **random session key** (one-time pad) and uses this key with a symmetric algorithm to encrypt the message (in the figure, *Block cipher* represents an authenticated encryption[6] algorithm). At the same time, the *sender* encrypts the session key using the public key of the *receiver*. This operation is done using a public-key algorithm, e.g., RSA. Once the encryption is done, the *sender* can send the encrypted session key along with the encrypted message to the *receiver*. The *receiver* can decrypt the session key using his/her private key, and consequently decrypt the message.

This idea of combining together symmetric and asymmetric (public-key) encryption can be used to implement end-to-end encryption (**E2EE**). E2EE is a communication system that encrypts messages exchanged by two users with the property that only the two users can decrypt the message. End-to-end encryption has become quite popular in the last years in software, and particularly messaging systems, such as WhatsApp[7]. More generally, when you have software used by many users, end-to-end encryption can be used to protect information exchanged by users. Only the users can access (decrypt) exchanged information; even the administrator of the system is not able to access this data.

Build end-to-end encryption in PHP

We want to implement end-to-end encryption for a web application with user authentication. We will use zend-crypt 3.1.0 to implement our cryptographic schemas. This component of Zend Framework uses PHP's OpenSSL extension[8] for its cryptographic primitives.

The first step is to create public and private keys for each users. Typically, this step can be done when the user credentials are created. To generare the pairs of keys, we can use `Zend\Crypt\PublicKey\RsaOptions`. Below is an example demonstrating how to generate public and private keys to store in the filesystem:

```
use Zend\Crypt\PublicKey\RsaOptions;
use Zend\Crypt\BlockCipher;

$username = 'alice';
$password = 'test'; // user's password

// Generate public and private key
$rsaOptions = new RsaOptions();
$rsaOptions->generateKeys([
    'private_key_bits' => 2048
]);
$publicKey  = $rsaOptions->getPublicKey()->toString();
$privateKey = $rsaOptions->getPrivateKey()->toString();

// store the public key in a .pub file
file_put_contents($username . '.pub', $publicKey);

// encrypt and store the private key in a file
$blockCipher = BlockCipher::factory('openssl', array('algo' => 'aes'));
$blockCipher->setKey($password);
file_put_contents($username, $blockCipher->encrypt($privateKey));
```

In the above example, we generated a private key of 2048 bits. If you are wondering why not 4096 bits, this is questionable and depends on the real use case. For the majority of applications, 2048 is still a good key size, at least until 2030. If you want more security and you don't care about the additional CPU time, you can increase the key size to 4096. I suggest reading the following blog posts for more information on key key size:

- RSA Key Sizes: 2048 or 4096 bits?: https://danielpocock.com/rsa-key-sizes-2048-or-4096-bits
- The Big Debate, 2048 vs. 4096, Yubico’s Position: https://www.yubico.com/2015/02/big-debate-2048-4096-yubicos-stand/
- HTTPS Performance, 2048-bit vs 4096-bit: https://blog.nytsoi.net/2015/11/02/nginx-https-performance

> In the example above, we did not generate the private key using a passphrase; this is because the OpenSSL extension of PHP does not support **AEAD** (Authenticated Encrypt with Associated Data) mode for ciphers yet, which is required in order to use passphrases.

The default passphrase encryption algorithm for OpenSSL is des-ede3-cbc[10] using PBKDF2[11] with 2048 iterations for generating the encryption key from the user's password. Even if this encryption algorithm is quite good, the number of iterations of PBKDF2 is not optimal; zend-crypt improves on this in a variety of ways, out-of-the-box. As demonstrated above, I use `Zend\Crypt\BlockCipher` to encrypt the private key; this class provides encrypt-then-authenticate[12] using the **AES-256** algorithm for encryption and **HMAC-SHA-256** for authentication. Moreover, `BlockCipher` uses the PBKDF2[11] algorithm to derivate the encryption key from the user's key (password). The default number of iterations for PBKDF2 is 5000, and you can increase it using the `BlockCipher::setKeyIteration()` method.

In the example, I stored the public and private keys in two files named, respectively, `$username.pub` and `$username`. Because the private file is encrypted, using the user's password, it can be access only by the user. This is a very important aspect for the security of the entire system (we take for granted that the web application stores the hashes of the user's passwords using a secure algorithm such as bcrypt[13]).

Once we have the public and private keys for the users, we can start using the hybrid cryptosystem provided by zend-crypt. For instance, imagine *Alice* wants to send an encrypted message to *Bob*:

```php
use Zend\Crypt\Hybrid;
use Zend\Crypt\BlockCipher;

$sender   = 'alice';
$receiver = 'bob';
$password = 'test'; // bob's password

$msg = sprintf('A secret message from %s!', $sender);

// encrypt the message using the public key of the receiver
$publicKey  = file_get_contents($receiver . '.pub');
$hybrid     = new Hybrid();
$ciphertext = $hybrid->encrypt($msg, $publicKey);

// send the ciphertext to the receiver

// decrypt the private key of bob
$blockCipher = BlockCipher::factory('openssl', ['algo' => 'aes']);
$blockCipher->setKey($password);
$privateKey = $blockCipher->decrypt(file_get_contents($receiver));

$plaintext = $hybrid->decrypt($ciphertext, $privateKey);

printf("%s\n", $msg === $plaintext ? "The message is: $msg" : 'Error!');
```

The above example demonstrates encrypting information between two users. Of course, in this case, the sender (*Alice*) knows the message because she wrote it. More in general, if we need to store a secret between multiple users, we need to specify the public keys to be used for encryption.

The hybrid component of zend-crypt supports encrypting messages for multiple recipients. To do so, pass an array of public keys in the `$publicKey` parameter of `Zend\Crypt\Hybrid::encrypt($data, $publicKey)`.

Below demonstrates encrypting a file for two users, *Alice* and *Bob*.

```php
use Zend\Crypt\Hybrid;
use Zend\Crypt\BlockCipher;

$data    = file_get_contents('path/to/file/to/protect');
$pubKeys = [
  'alice' => file_get_contents('alice.pub'),
  'bob'   => file_get_contents('bob.pub')
];

$hybrid     = new Hybrid();

// Encrypt using the public keys of both alice and bob
$ciphertext = $hybrid->encrypt($data, $pubKeys);

file_put_contents('file.enc', $ciphertext);

$blockCipher = BlockCipher::factory('openssl', ['algo' => 'aes']);

$passwords = [
  'alice' => 'password of Alice',
  'bob'   => 'password of Bob'
];

// decrypt using the private keys of alice and bob, one at time
foreach ($passwords as $id => $pass) {
  $blockCipher->setKey($pass);
  $privateKey = $blockCipher->decrypt(file_get_contents($id));
  $plaintext  = $hybrid->decrypt($ciphertext, $privateKey, null, $id);
  printf("%s for %s\n", $data === $plaintext ? 'Decryption ok' : 'Error', $id);
}
```

For decryption, I used a hard coded password for the users. Usually, the user's password is provided during the login process of a web application and should not be stored as permanent data; for instance, the user's password can be saved in a PHP session variable for temporary usage. If you use sessions to save the user's password, ensure that data is protected; the PHP-Secure-Session[14] library or the Suhosin[15] PHP extension will help you do so.

To decrypt the file, I used the `Zend\Crypt\Hybrid::decrypt()` method, where I specified the `$privateKey`, a `null` passphrase, and finally the `$id` of the privateKey. This parameters are necessary to find the correct key to use in the header of the encrypted message.

Footnotes

1. https://github.com/zendframework/zend-crypt ↩

2. https://docs.zendframework.com/zend-crypt/hybrid/ ↩

3

3. https://en.wikipedia.org/wiki/End-to-end_encryption ↩

4. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard ↩

5. https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29 ↩

6. https://en.wikipedia.org/wiki/Authenticated_encryption ↩

7. https://www.whatsapp.com/faq/en/general/28030015 ↩

8. http://php.net/manual/en/book.openssl.php ↩

9. https://wiki.php.net/rfc/openssl_aead ↩

10. https://en.wikipedia.org/wiki/Triple_DES ↩

11. https://en.wikipedia.org/wiki/PBKDF2 ↩

12. http://www.daemonology.net/blog/2009-06-24-encrypt-then-mac.html ↩

13. https://en.wikipedia.org/wiki/Bcrypt ↩

14. https://github.com/ezimuel/PHP-Secure-Session ↩

15. https://suhosin.org ↩

Create ZPKs the Easy Way

by Enrico Zimuel

Zend Server[1] provides the ability to deploy applications to a single server or cluster of servers via the ZPK[2] package format. We offer the package zfcampus/zf-deploy[3] for creating ZPK packages from Zend Framework and Apigility applications, but how can you create these for Expressive, or, really, *any* PHP application?

Requirements

To create the ZPK, you need a few things:

- The `zip` binary. ZPKs are ZIP files with specific artifacts.
- The `composer` binary, so you can install dependencies.
- An `.htaccess` file, if your Zend Server installation is using Apache.
- A `deployment.xml` file.

htaccess

If you are using Apache, you'll want to make sure that you setup things like rewrite rules for your application. While this can be done when defining the vhost in the Zend Server admin UI, using an `.htaccess` file makes it easier to make changes to the rules between deployments.

The following `.htaccess` file will work for many (most?) PHP projects. Place it relative to your project's front controller script; in the case of Expressive, Zend Framework, and Apigility, that would mean `public/index.php`, and thus `public/.htaccess`:

```
RewriteEngine On
# The following rule tells Apache that if the requested filename
# exists, simply serve it.
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]

# The following rewrites all other queries to index.php. The
# condition ensures that if you are using Apache aliases to do
# mass virtual hosting, the base path will be prepended to
# allow proper resolution of the index.php file; it will work
# in non-aliased environments as well, providing a safe, one-size
# fits all solution.
RewriteCond %{REQUEST_URI}::$1 ^(/.+)(.+)::\2$
RewriteRule ^(.*) - [E=BASE:%1]
RewriteRule ^(.*)$ %{ENV:BASE}index.php [NC,L]
```

deployment.xml

The `deployment.xml` tells Zend Server about the application you are deploying. What is listed below will work for Expressive, Zend Framework, and Apigility applications, and likely a number of other PHP applications. The main things to pay attention to are:

- The `name` should typically match the application name you've setup in Zend Server.
- The `version.release` value should be updated for each release; this allows you to use rollback features.
- The `appdir` value is the project root. An empty value indicates the same directory as the `deployment.xml` lives in.
- The `docroot` value is the directory from which the vhost will serve files.

So, as an example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<package version="2.0" xmlns="http://www.zend.com/server/deployment-descriptor/1.0">
    <type>application</type>
    <name>API</name>
    <summary>API for all the things!</summary>
    <version>
        <release>1.0</release>
    </version>
    <appdir></appdir>
    <docroot>public</docroot>
</package>
```

Installing dependencies

When you're ready to build a package, you should install your dependencies. However, don't install them any old way; install them in a production-ready way. This means:

- Specifying that composer optimize the autoloader (`--optimize-autoloader`).
- Use production dependencies only (`--no-dev`).
- Prefer distribution packages (versus source installs) (`--prefer-dist`).

So:

```
$ composer install --no-dev --prefer-dist --optimize-autoloader
```

Create the ZPK

Finally, we can now create the ZPK, using the `zip` command:

```
$ zip -r api-1.0.0.zpk . -x api-1.0.0.zpk -x '*.git/*'
```

This creates the file `api-1.0.0.zpk` with all contents of the current directory minus the `.git` directory and the ZPK itself (these are excluded via the `-x` flags). (You may want/need to specify additional exclusions; the above are typical, however.)

You can then upload the ZPK to the web interface, or use the Zend Server SDK[4].

Simple example: single-directory PoC

Let's say you want to do a proof-of-concept, and will be creating an `index.php` in the project root to test out an idea. You would use the above `.htaccess` , but keep it in the project root. Your `deployment.xml` would look the same, except that the `docroot` value would be empty:

```xml
<?xml version="1.0" encoding="utf-8"?>
<package version="2.0" xmlns="http://www.zend.com/server/deployment-descriptor/1.0">
    <type>application</type>
    <name>POC</name>
    <summary>Proof-of-concept of a very cool idea</summary>
    <version>
        <release>0.1.0</release>
    </version>
    <appdir></appdir>
    <docroot></docroot>
</package>
```

You'd then run:

```
$ zip -r poc-0.1.0.zpk . -x poc-0.1.0.zpk
```

Done!

Fin

ZPKs make creating and staging deployment packages fairly easy — once you know how to create the packages. We hope that this post helps demystify the first steps in creating a ZPK for your application.

Visit the Zend Server documentation[5] for more information on ZPK structure.

Footnotes

> [1]. http://www.zend.com/en/products/zend_server ↩

> [2]. http://files.zend.com/help/Zend-Server/content/application_package.htm ↩

> [3]. https://github.com/zfcampus/zf-deploy ↩

> [4]. https://github.com/zend-patterns/ZendServerSDK ↩

> [5]. http://files.zend.com/help/Zend-Server/content/understanding_the_application_package_structure.htm ↩

Using Laravel Homestead with Zend Framework Projects

by Enrico Zimuel

Laravel Homestead[1] is an interesting project by the Laravel community that provides a Vagrant[2] box for PHP developers. It includes a full set of services for PHP developers, such as the Nginx web server, PHP 7.1, MySQL, Postgres, Redis, Memcached, Node, and more.

One the most interesting features of this project is the ability to enable it per project. This means you can run a vagrant box for your specific PHP project.

In this article, we'll examine using it for Zend Framework MVC, Expressive, and Apigility projects. In each case, installation and usage is exactly the same.

Install the Vagrant box

The first step is to install the laravel/homestead[3] vagrant box. This box works with a variety of providers: VirtualBox 5.1[4], VMWare[5], or Parallels[6].

We used VirtualBox and the following command to install the laravel/homestead box:

```
$ vagrant box add laravel/homestead
```

The box is 981 MB, so it will take some minutes to download.

Homestead, by default, uses the host name `homestead.app`, and requires that you update your system hosts file to point that domain to the virtual machine IP address. To faciliate that, Homestead provides integration with the vagrant-hostsupdater[7] Vagrant plugin. We recommend installing that before your initial run of the virtual machine:

```
$ vagrant plugin install vagrant-hostsupdater
```

Use Homestead in ZF projects

Once you have installed the laravel/homestead vagrant box, you can use it globally or per project.

If we install Homestead per-project, we will have a full development server configured directly in the local folder, without sharing services with other projects. This is a big plus!

To use Homestead per-project, we need to install the laravel/homestead[8] package within our Zend Framework, Apigility, or Expressive project. This can be done using Composer[9] with the following command:

```
$ composer require --dev laravel/homestead
```

After installation, execute the `homestead` command to build the `Vagrantfile`:

```
$ vendor/bin/homestead make
```

This command creates both the `VagrantFile` and a `Homestead.yaml` configuration file.

Configuring Homestead

By default, the vagrant box is set up at address `192.168.10.10` with the hostname `homestead.app`. You can change the IP address in `Homestead.yaml` if you want, as well as the hostname (via the `sites[].map` key).

The `Homestead.yaml` configuration file contains all details about the vagrant box configuration. The following is an example:

```
---
ip: "192.168.10.10"
memory: 2048
cpus: 1
hostname: expressive-homestead
name: expressive-homestead
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub

keys:
    - ~/.ssh/id_rsa

folders:
    - map: "/home/enrico/expressive-homestead"
      to: "/home/vagrant/expressive-homestead"

sites:
    - map: homestead.app
      to: "/home/vagrant/expressive-homestead/public"

databases:
    - homestead
```

This configuration file is very simple and intuitive; for instance, the folders to be used are reported in the `folders` section; the `map` value is the local folder of the project, the `to` value is the folder on the virtual machine.

If you want to add or change more features in the virtual machine you can used the `Homestead.yaml` configuration file. For instance, if you prefer to add MariaDB instead of MySQL, you need to add the `mariadb` option:

```
ip: "192.168.10.10"
memory: 2048
cpus: 1
hostname: expressive-homestead
name: expressive-homestead
provider: virtualbox
mariadb: true
```

This option will remove MySQL and install MariaDB.

> ### SSH keys managed by GPG
>
> One of our team uses the gpg-agent as an ssh-agent, which caused some configuration problems initially, as the `~/.ssh/id_rsa` and its `.pub` sibling were not present.
>
> When using gpg-agent for serving SSH keys, you can export the key using `ssh-add -L`. This may list several keys, but you should be able to find the correct one. Copy it to the file `~/.ssh/gpg_key.pub`, and then copy that file to `~/.ssh/gpg_key.pub.pub`. Update the `Homestead.yaml` file to reflect these new files:
>
> ```
> authorize: ~/.ssh/gpg_key.pub.pub
> keys:
>     - ~/.ssh/gpg_key.pub
> ```
>
> The gpg-agent will take care of sending the appropriate key from there.

Running Homestead

To run the vagrant box, execute the following within your project root:

```
$ vagrant up
```

If you open a browser to `http://homestead.app` you should now see your application running.

> ### Manually managing your hosts file
>
> If you chose not to use vagrant-hostsupdater, you will need to update your system hosts file.
>
> On Linux and Mac, update the `/etc/hosts` file to add the following line:
>
> ```
> 192.168.10.10 homestead.app
> ```
>
> On Windows, the host file is located in `C:\Windows\System32\drivers\etc\hosts`.

More information

We've tested this setup with each of the Zend Framework zend-mvc skeleton application, Apigility, and Expressive, and found the setup "just worked"! We feel it provides excellent flexibility in setting up development environments, giving developers a wide range of tools and technologies to work with as they develop applications.

For more information about Laravel Homestead, visit the official documentation[10] of the project.

Footnotes

> [1]. https://laravel.com/docs/5.4/homestead ↩
>
> [2]. https://www.vagrantup.com/ ↩
>
> [3]. https://atlas.hashicorp.com/laravel/boxes/homestead ↩
>
> [4]. https://www.virtualbox.org/wiki/Downloads ↩
>
> [5]. https://www.vmware.com/ ↩
>
> [6]. http://www.parallels.com/products/desktop/ ↩
>
> [7]. https://github.com/cogitatio/vagrant-hostsupdater ↩
>
> [8]. https://github.com/laravel/homestead ↩
>
> [9]. https://getcomposer.org/ ↩
>
> [10]. https://laravel.com/docs/5.4/homestead ↩

Copyright note

Rogue Wave helps thousands of global enterprise customers tackle the hardest and most complex issues in building, connecting, and securing applications. Since 1989, our platforms, tools, components, and support have been used across financial services, technology, healthcare, government, entertainment, and manufacturing, to deliver value and reduce risk. From API management, web and mobile, embeddable analytics, static and dynamic analysis to open source support, we have the software essentials to innovate with confidence.

- https://www.roguewave.com/