RogueWave SOFTWARE

zend®

EXPRESSIVE COOKBOOK

A collection of PHP recipes
for PSR-7 middleware applications

Table of Contents

Expressive Cookbook

This book contains a collection of articles on Expressive[1], a PSR-7[2] microframework for building *middleware* applications in PHP. It collects most of the articles on PSR-7 and Expressive published in 2017 by Matthew Weier O'Phinney and Enrico Zimuel on the official Zend Framework blog[3].

The goal of this book is to guide PHP developers in the usage of Expressive. We feel middleware is an elegant way to write web applications as the approach allows you to write targeted, single-purpose code for interacting with an HTTP request in order to produce an HTTP response. Each middleware should do exactly one thing, and developers should model complex workflows by stacking middleware. In this book, we demonstrate a number of recipes that demonstrate this, and our goal is to help you, the developer, gain mastery of the middleware patterns.

We think that PSR-7 and middleware represent the future of web development in PHP, from small to complex enterprise projects.

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

Links

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

> [2]. http://www.php-fig.org/psr/psr-7/ ↩

> [3]. https://framework.zend.com/blog ↩

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/

Specialized Response Implementations in Diactoros

By Matthew Weier O'Phinney

When writing PSR-7[1] middleware, at some point you'll need to return a response.

Maybe you'll be returning an empty response, indicating something along the lines of successful deletion of a resource. Maybe you need to return some HTML, or JSON, or just plain text. Maybe you need to indicate a redirect.

But here's the problem: a *generic* response typically has a very *generic* constructor. Take, for example, `Zend\Diactoros\Response`:

```
public function __construct(
    $body = 'php://memory',
    $status = 200,
    array $headers = []
)
```

`$body` in this signature allows either a `Psr\Http\Message\StreamInterface` instance, a PHP resource, or a string identifying a PHP stream. This means that it's not terribly easy to create even a simple HTML response!

> To be fair, there are good reasons for a generic constructor: it allows setting the initial state in such a way that you'll have a fully populated instance immediately. However, the means for doing so, in order to be generic, leads to convoluted code for most consumers.

Fortunately, Diactoros provides a number of convenience implementations to help simplify the most common use cases.

EmptyResponse

The standard response from an API for a successful deletion is generally a `204 No Content`. Sites emitting webhook payloads often expect a `202 Accepted` with no content. Many APIs that allow creation of resources will return a `201 Created`; these may or may not have content, depending on implementation, with some being empty, but returning a `Location` header with the URI of the newly created resource.

Clearly, in such cases, if you don't need content, why would you be bothered to create a stream? To answer this, we have `Zend\Diactoros\Response\EmptyResponse` , with the following constructor:

```
public function __construct($status = 204, array $headers = [])
```

So, a `DELETE` endpoint might return this on success:

```
return new EmptyResponse();
```

A webhook endpoint might do this:

```
return new EmptyResponse(StatusCodeInterface::STATUS_ACCEPTED);
```

An API that just created a resource might do the following:

```
return new EmptyResponse(
    StatusCodeInterface::STATUS_CREATED,
    ['Location' => $resourceUri]
);
```

RedirectResponse

Redirects are common within web applications. We may want to redirect a user to a login page if they are not currently logged in; we may have changed where some of our content is located, and redirect users requesting the old URIs; etc.

`Zend\Diactoros\Response\RedirectResponse` provides a simple way to create and return a response indicating an HTTP redirect. The signature is:

```
public function __construct($uri, $status = 302, array $headers = [])
```

where `$uri` may be either a string URI, or a `Psr\Http\Message\UriInterface` instance. This value will then be used to seed a `Location` HTTP header.

```
return new RedirectResponse('/login');
```

You'll note that the `$status` defaults to 302. If you want to set a permanent redirect, pass `301` for that argument:

```
return new RedirectResponse('/archives', 301);

// or, using fig/http-message-util:
return new RedirectResponse('/archives', StatusCodeInterface::STATUS_PERMANENT_REDIREC
T);
```

Sometimes you may want to set an header as well; do that by passing the third argument, an array of headers to provide:

```
return new RedirectResponse(
    '/login',
    StatusCodeInterface::STATUS_TEMPORARY_REDIRECT,
    ['X-ORIGINAL_URI' =>  $uri->getPath()]
);
```

TextResponse

Sometimes you just want to return some text, whether it's plain text, XML, YAML, etc. When doing that, taking the extra step to create a stream feels like overhead:

```
$stream = new Stream('php://temp', 'wb+');
$stream->write($content);
```

To simplify this, we offer `Zend\Diactoros\Response\TextResponse`, with the following signature:

```
public function __construct($text, $status = 200, array $headers = [])
```

By default, it will use a `Content-Type` of `text/plain`, which means you'll often need to supply a `Content-Type` header with this response.

Let's return some plain text:

```
return new TextResponse('Hello, world!');
```

Now, let's try returning a Problem Details XML response:

```
return new TextResponse(
    $xmlPayload,
    StatusCodeInterface::STATUS_UNPROCESSABLE_ENTITY,
    ['Content-Type' => 'application/problem+xml']
);
```

If you have some textual content, this is the response for you.

HtmlResponse

The most common response from web applications is HTML. If you're returning HTML, even the `TextResponse` may seem a bit much, as you're forced to provide the `Content-Type` header. To answer that, we provide `Zend\Diactoros\Response\HtmlResponse`, which is exactly the same as `TextResponse`, but with a default `Content-Type` header specifying `text/html; charset=utf-8` instead.

As an example:

```
return new HtmlResponse($renderer->render($template, $view));
```

JsonResponse

For web APIs, JSON is generally the *lingua franca*. Within PHP, this generally means passing an array or object to `json_encode()`, and supplying a `Content-Type` header of `application/json` or `application/{type}+json`, where `{type}` is a more specific mediatype.

Like text and HTML, you likely don't want to do this manually every time:

```
$json = json_encode(
  $data,
  JSON_HEX_TAG | JSON_HEX_APOS | JSON_HEX_QUOT | JSON_UNESCAPED_SLASHES
);
$stream = new Stream('php://temp', 'wb+');
$stream->write($json);
$response = new Response(
    $stream,
    StatusCodeInterface::STATUS_OK,
    ['Content-Type' => 'application/json']
);
```

To simplify this, we provide `Zend\Diactoros\Response\JsonResponse`, with the following constructor signature:

```
public function __construct(
    $data,
    $status = 200,
    array $headers = [],
    $encodingOptions = self::DEFAULT_JSON_FLAGS
) {
```

where `$encodingOptions` defaults to the flags specified in the previous example.

This means our most common use case now becomes this:

```
return new JsonResponse($data);
```

What if we want to return a JSON-formatted Problem Details response?

```
return new JsonResponse(
    $details,
    StatusCodeInterface::STATUS_UNPROCESSABLE_ENTITY,
    ['Content-Type' => 'application/problem+json']
);
```

One common workflow we've seen with JSON responses is that developers often want to manipulate them on the way out through middleware. As an example, they may want to add additional `_links` elements to HAL responses, or add counts for collections.

Starting in version 1.5.0, we provide a few extra methods on this particular response type:

```
public function getPayload() : mixed;
public function getEncodingOptions() : int;
public function withPayload(mixed $data) : JsonResponse;
public function withEncodingOptions(int $options) : JsonResponse;
```

Essentially, what happens is we now store not only the encoded `$data` internally, but the raw data; this allows you to pull it, manipulate it, and then create a new instance with the updated data. Additionally, we allow specifying a different set of encoding options later; this can be useful, for instance, for adding the `JSON_PRETTY_PRINT` flag when in development. When the options are changed, the new instance will also re-encode the existing data.

First, let's look at altering the payload on the way out. zend-expressive-hal injects `_total_items`, `_page`, and `_page_count` properties, and you may want to remove the underscore prefix for each of these:

```php
function (ServerRequestInterface $request, DelegateInterface $delegate) : ResponseInte
rface
{
    $response = $delegate->process($request);
    if (! $response instanceof JsonResponse) {
        return $response;
    }

    $payload = $response->getPayload();
    if (! isset($payload['_total_items'])) {
        return $response;
    }

    $payload['total_items'] = $payload['_total_items'];
    unset($payload['_total_items']);

    if (isset($payload['_page'])) {
        $payload['page'] = $payload['_page'];
        $payload['page_count'] = $payload['_page_count'];
        unset($payload['_page'], $payload['_page_count']);
    }

    return $response->withPayload($payload);
}
```

Now, let's write middleware that sets the `JSON_PRETTY_PRINT` option when in development mode:

```php
function (
    ServerRequestInterface $request,
    DelegateInterface $delegate
) : ResponseInterface use ($isDevelopmentMode) {
    $response = $delegate->process($request);

    if (! $isDevelopmentMode || ! $response instanceof JsonResponse) {
        return $response;
    }

    $options = $response->getEncodingOptions();
    return $response->withEncodingOptions($options | JSON_PRETTY_PRINT);
}
```

These features can be really powerful when shaping your API!

Summary

The goal of PSR-7 is to provide the ability to standardize on interfaces for your HTTP interactions. However, at some point you need to choose an actual implementation, and your choice will often be shaped by the *features* offered, particularly if they provide convenience in your development process. Our goal with these various custom response implementations is to provide *convenience* to developers, allowing them to focus on what they need to return, not *how* to return it.

You can check out more in the Diactoros documentation[2].

Footnotes

> [1]. http://www.php-fig.org/psr/psr-7/ ↩

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

Emitting Responses with Diactoros

By Matthew Weier O'Phinney

When writing middleware-based applications, at some point you will need to *emit your response*.

PSR-7[1] defines the various interfaces related to HTTP messages, but does not define how they will be used. Diactoros[2] defines several utility classes for these purposes, including a `ServerRequestFactory` for generating a `ServerRequest` instance from the PHP SAPI in use, and a set of *emitters*, for emitting responses back to the client. In this post, we'll detail the purpose of emitters, the emitters shipped with Diactoros, and some strategies for emitting content to your users.

What is an emitter?

In vanilla PHP applications, you might call one or more of the following functions in order to provide a response to your client:

- `http_response_code()` for emitting the HTTP response code to use; this must be called before any output is emitted.
- `header()` for emitting response headers. Like `http_response_code()`, this must be called before any output is emitted. It may be called multiple times, in order to set multiple headers.
- `echo()`, `printf()`, `var_dump()`, and `var_export()` will each emit output to the current output buffer, or, if none is present, directly to the client.

One aspect PSR-7 aims to resolve is the ability to generate a response piece-meal, including adding content and headers in whatever order your application requires. To accomplish this, it provides a `ResponseInterface` with which your application interacts, and which aggregates the response status code, its headers, and all content.

Once you have a complete response, however, you need to emit it.

Diactoros provides *emitters* to solve this problem. Emitters all implement `Zend\Diactoros\Response\EmitterInterface`:

```php
namespace Zend\Diactoros\Response;

use Psr\Http\Message\ResponseInterface;

interface EmitterInterface
{
    /**
     * Emit a response.
     *
     * Emits a response, including status line, headers, and the message body,
     * according to the environment.
     *
     * Implementations of this method may be written in such a way as to have
     * side effects, such as usage of header() or pushing output to the
     * output buffer.
     *
     * Implementations MAY raise exceptions if they are unable to emit the
     * response; e.g., if headers have already been sent.
     *
     * @param ResponseInterface $response
     */
    public function emit(ResponseInterface $response);
}
```

Diactoros provides two emitter implementations, both geared towards standard PHP SAPI implementations:

- `Zend\Diactoros\Emitter\SapiEmitter`
- `Zend\Diactoros\Emitter\SapiStreamEmitter`

Internally, they operate very similarly: they emit the response status code, all headers, and the response body content. Prior to doing so, however, they check for the following conditions:

- Headers have not yet been sent.
- If any output buffers exist, no content is present.

If either of these conditions is not true, the emitters raise an exception. This is done to ensure that consistent content can be emitted; mixing PSR-7 and global output leads to unexpected and inconsistent results. If you are using middleware, use things like the error log, loggers, etc. if you want to debug, instead of mixing strategies.

Emitting files

As noted above, one of the two emitters is the `SapiStreamEmitter`. The normal `SapiEmitter` emits the response body at once via a single `echo` statement. This works for most general markup and JSON payloads, but when returning files (for example, when providing file downloads via your application), this strategy can quickly exhaust the amount of memory PHP is allowed to consume.

The `SapiStreamEmitter` is designed to answer the problem of file downloads. It emits a chunk at a time (8192 bytes by default). While this can mean a bit more performance overhead when emitting a large file, as you'll have more method calls, it also leads to reduced *memory* overhead, as less content is in memory at any given time.

The `SapiStreamEmitter` has another important feature, however: it allows sending *content ranges*.

Clients can opt-in to receiving small chunks of a file at a time. While this means more network calls, it can also help prevent corruption of large files by allowing the client to re-try failed requests in order to stitch together the full file. Doing so also allows providing progress status, or even buffering streaming content.

When requesting content ranges, the client will pass a `Range` header:

```
Range: bytes=1024-2047
```

It is up to the server then to detect such a header and return the requested range. Servers indicate that they are doing so by responding with a `Content-Range` header with the range of bytes being returned and the total number of bytes possible; the response body then only contains those bytes.

```
Content-Range: bytes=1024-2047/11576
```

As an example, middleware that allows returning a content range might look like the following:

```
function (ServerRequestInterface $request, DelegateInterface $delegate) : ResponseInte
rface
{
    $stream = new Stream('path/to/download/file', 'r');
    $response = new Response($stream);

    $range = $request->getHeaderLine('range');
    if (empty($range)) {
        return $response;
    }

    $size  = $body->getSize();
    $range = str_replace('=', ' ', $range);
    $range .= '/' . $size;

    return $response->withHeader('Content-Range', $range);
}
```

> You'll likely want to validate that the range is within the size of the file, too!

The above code emits a `Content-Range` response header if a `Range` header is in the request. However, how do we ensure *only* that range of bytes is emitted?

By using the `SapiStreamEmitter`! This emitter will detect the `Content-Range` header and use it to read and emit only the bytes specified by that header; no extra work is necessary!

Mixing and matching emitters

The `SapiEmitter` is perfect for content generated within your application — HTML, JSON, XML, etc. — as such content is usually of reasonable length, and will not exceed normal memory and resource limits.

The `SapiStreamEmitter` is ideal for returning file downloads, but can lead to performance overhead when emitting standard application content.

How can you mix and match the two?

Expressive answers this question by providing `Zend\Expressive\Emitter\EmitterStack`. The class acts as a stack (last in, first out), executing each emitter composed until one indicates it has handled the response.

This class capitalizes on the fact that the return value of `EmitterInterface` is undefined. Emitters that return a boolean `false` indicate they were *unable to handle the response*, allowing the `EmitterStack` to move to the next emitter in the stack. The first emitter to return a non-`false` value halts execution.

Both the emitters defined in zend-diactoros return `null` by default. So, if we want to create a stack that first tries `SapiStreamEmitter`, and then defaults to `SapiEmitter`, we could do the following:

```
use Psr\Http\Message\ResponseInterface;
use Zend\Diactoros\Response\EmitterInterface;
use Zend\Diactoros\Response\SapiEmitter;
use Zend\Diactoros\Response\SapiStreamEmitter;
use Zend\Expressive\Emitter\EmitterStack;

$emitterStack = new EmitterStack();
$emitterStack->push(new SapiEmitter());
$emitterStack->push(new class implements EmitterInterface {
    public function emit(ResponseInterface $response)
    {
        $contentSize = $response->getBody()->getSize();

        if ('' === $response->getHeaderLine('content-range')
            && $contentSize < 8192
        ) {
            return false;
        }

        $emitter = new SapiStreamEmitter();
        return $emitter->emit($response);
    }
});
```

The above will execute our anonymous class as the first emitter. If the response has a `Content-Range` header, or if the size of the content is greater than 8k, it will use the `SapiStreamEmitter`; otherwise, it returns `false`, allowing the next emitter in the stack, `SapiEmitter`, to execute. Since that emitter always returns null, it acts as a default emitter implementation.

In Expressive, if you were to wrap the above in a factory that returns the `$emitterStack`, and assign that factory to the `Zend\Diactoros\Emitter\EmitterInterface` service, then the above stack will be used by `Zend\Expressive\Application` for the purpose of emitting the application response!

Summary

Emitters provide you the ability to return the response you have aggregated in your application to the client. They are intended to have side-effects: sending the response code, response headers, and body content. Different emitters can use different strategies when emitting responses, from simply `echo`ing content, to iterating through chunks of content (as the `SapiStreamEmitter` does). Using Expressive's `EmitterStack` can provide you with a way to select different emitters for specific response criteria.

For more information:

- Read the Diactoros emitter documentation: https://docs.zendframework.com/zend-diactoros/emitting-responses/
- Read the Expressive emitter documentation: https://docs.zendframework.com/zend-expressive/features/emitters/

Footnotes

> 1. http://www.php-fig.org/psr/psr-7/ ↩

> 2. https://docs.zendframework.org/zend-diactoros/ ↩

Migrating to Expressive 2.0

by Matthew Weier O'Phinney

Zend Expressive 2 was released in March 2017[1]. A new major version implies breaking changes, which often poses a problem when migrating. That said, we did a lot of work behind the scenes to try and ensure that migrations can happen without too much effort, including providing migration tools to ease the transition.

In this tutorial, we will detail migrating an existing Expressive application from version 1 to version 2.

> ### How we tested this
>
> We used Adam Culp's expressive-blastoff[2] repository as a test-bed for this tutorial, and you can follow along from there if you want, by checking out the 1.0 tag of that repository:
>
> ```
> $ git clone https://github.com/adamculp/expressive-blastoff
> $ cd expressive-blastoff
> $ git checkout 1.0
> $ composer install
> ```
>
> We have also successfully migrated a number of other applications, including the Zend Framework website itself, using essentially the same approach. As is the case with any such tutorial, your own experience may vary.

Updating dependencies

First, create a new feature branch for the migration, to ensure you do not clobber working code. If you are using git, this might look like this:

```
$ git checkout -b feature/expressive-2
```

If you have not yet installed dependencies, we recommend doing so:

```
$ composer install
```

Now, we'll update dependencies to get Expressive 2. Doing so on an existing project requires a number of other updates as well:

- You will need to update whichever router implementation you use, as we have released new major versions of all routers, to take advantage of a new major version of the zend-expressive-router `RouterInterface`. You can pin these to `^2.0`.
- You will need to update the zend-expressive-helpers package, as it now also depends on the new `RouterInterface` changes. You can pin this to `^3.0`.
- You will need to update your template renderer, if you have one installed. These received minor version bumps in order to add compatibility with the new zend-expressive-helpers release; however, since we'll be issuing a `require` statement to upgrade Expressive, we need to specify the new template renderer version as well. Constraints for the supported renderers are:
    - `zendframework/zend-expressive-platesrenderer:^1.2`
    - `zendframework/zend-expressive-twigrenderer:^1.3`
    - `zendframework/zend-expressive-zendviewrenderer:^1.3`

As an example, if you are using the recommended packages zendframework/zend-expressive-fastroute and zendframework/zend-expressive-platesrenderer, you will update to Expressive 2.0 using the following statement:

```
$ composer update --with-dependencies "zendframework/zend-expressive:^2.0" \
> "zendframework/zend-expressive-fastroute:^2.0" \
> "zendframework/zend-expressive-helpers:^3.0" \
> "zendframework/zend-expressive-platesrenderer:^1.2"
```

At this point, try out your site. In many cases, it *should* continue to "just work."

> ## Common errors
>
> We say *should* for a reason. There are a number of features that will not work, but were not commonly used by end-users, including accessing properties on the request/response decorators that Stratigility 1 shipped (on which Expressive 1 was based), and usage of Stratigility 1 "error middleware" (which was removed in the version 2 releases). While these were documented, many users were not aware of the features and/or did not use them. If you did, however, you will notice your site will not run following the upgrade. Don't worry; we cover tools that will solve these issues in the next section!

Migration

At this point, there's a few more steps you should take to fully migrate your application; in some cases, your application is currently broken, and will require these changes to work in the first place!

We provide CLI tooling that assists in these migrations via the package zendframework/zend-expressive-tooling. Add this as a development requirement to your application now:

```
$ composer require --dev --update-with-dependencies zendframework/zend-expressive-tool
ing
```

(The `--update-with-dependencies` may be necessary to pick up newer versions of zend-stdlib and zend-code, among others.)

Expressive 1 was based on Stratigility 1, which decorated the request and response objects with wrappers that provide access to the original incoming request, URI, and response. With Stratigility 2 and Expressive 2, these decorators have been removed; however access to these artifacts is available via request attributes. As such, we provide a tool to scan for usage of these and fix them when possible. Let's invoke it now:

```
$ ./vendor/bin/expressive-migrate-original-messages scan
```

(If your code is in a directory other than `src/` , then use the `--help` switch for options on specifying that directory.)

Most likely the tool won't find anything. In some cases, it *will* find something, and try to correct it. The one thing it cannot correct are calls to `getOriginalResponse()` ; in such cases, the tool details how to correct those problems, and in what files they occur.

Next, we'll scan for legacy error middleware. This was middleware defined in Stratigility with an alternate signature:

```
function (
    $error,
    ServerRequestInterface $request,
    ResponseInterface $response,
    callable $next
) : ResponseInterface
```

Such middleware was invoked by calling `$next` with a third argument:

```
$response = $next($request, $response, $error);
```

This style of middleware has been removed from Stratigility 2 and Expressive 2, and will not work at all. We provide another tool for finding both error middleware, as well as invocations of error middleware:

```
$ ./vendor/bin/expressive-scan-for-error-middleware scan
```

(If your code is in a directory other than `src/`, then use the `--help` switch for options on specifying that directory.)

This tool does not change any code, but it will tell you files that contain problems, and give you information on how to correct the issues.

Finally, we'll migrate to a *programmatic pipeline*. In Expressive 1, the skeleton defined the pipeline and routes via configuration. Many users have indicated that using the Expressive API tends to be easier to learn and understand than the configuration; additionally, IDEs and static analyzers are better able to determine if programmatic pipelines and routing are correct than configuration-driven ones.

As with the other migration tasks, we provide a tool for this:

```
$ ./vendor/bin/expressive-pipeline-from-config generate
```

This tool loads your existing configuration, and then does the following:

- Creates `config/autoload/programmatic-pipeline.global.php`, which contains directives to tell Expressive to ignore configured pipelines and routing, and defines dependencies for new error handling and pipeline middleware.
- Creates `config/pipeline.php` with your application middleware pipeline.
- Creates `config/routes.php` with your application routing definitions.
- Updates `public/index.php` to include the above two files prior to calling `$app->run()`.

The tool will also tell you if it encounters legacy error middleware in your configuration; if it does, it skips adding directives to compose it in the application pipeline, but notifies you it is doing so. Be aware of that, if you depended on the feature previously; in most cases, if you've been following this tutorial step-by-step, you've already eliminated them.

At this point, try out your application again! If all went well, this should "just work."

Bonus steps!

While the above will get your application migrated, V2 of the skeleton application offers three additional features that were not present in the original v1 releases:

- self-invoking function in `public/index.php` in order to prevent global variable declarations.
- ability to define and/or use middleware modules, via zend-config-aggregator.
- development mode.

Self-invoking function

The point of this change is to prevent addition of variables into the `$GLOBAL` scope. This is done by creating a *self-invoking function* around the directives in `public/index.php` that create and use variables.

After completing the earlier steps, you should have lines like the following in your `public/index.php`:

```
/** @var \Interop\Container\ContainerInterface $container */
$container = require 'config/container.php';

/** @var \Zend\Expressive\Application $app */
$app = $container->get(\Zend\Expressive\Application::class);
require 'config/pipeline.php';
require 'config/routes.php';
$app->run();
```

We'll create a self-invoking function around them. If you are using PHP 7+, this looks like the following:

```
(function () {
  /** @var \Interop\Container\ContainerInterface $container */
  $container = require 'config/container.php';

  /** @var \Zend\Expressive\Application $app */
  $app = $container->get(\Zend\Expressive\Application::class);
  require 'config/pipeline.php';
  require 'config/routes.php';
  $app->run();
})();
```

If you're still using PHP 5.6, you need to use `call_user_func()`:

```
call_user_func(function () {
  /** @var \Interop\Container\ContainerInterface $container */
  $container = require 'config/container.php';

  /** @var \Zend\Expressive\Application $app */
  $app = $container->get(\Zend\Expressive\Application::class);
  require 'config/pipeline.php';
  require 'config/routes.php';
  $app->run();
});
```

zend-config-aggregator

zendframework/zend-config-aggregator is at the heart of the modular middleware system[3]. It works as follows:

- *Modules* are just libraries or packages that define a `ConfigProvider` class. These classes are stateless and define an `__invoke()` method that returns an array of configuration.
- The `config/config.php` file then uses `Zend\ConfigAggregator\ConfigAggregator` to, well, *aggregate configuration* from a variety of sources, including `ConfigProvider` classes, as well as other specialized providers (e.g., PHP file provider for aggregating PHP configuration files, array provider for supplying hard-coded array configuration, etc.). This package provides built-in support for configuration caching as well.

We also provide a Composer plugin, zend-component-installer, that works with configuration files that utilize the `ConfigAggregator`. It executes during install operations, and checks the package being installed for configuration indicating it provides a `ConfigProvider`; if so, it will then prompt you, asking if you want to add it to your configuration. This is a great way to automate addition of dependencies and module-specific configuration to your application!

To get started, let's add zend-config-aggregator to our application:

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

We'll also add the `zend-component-installer`, but as a development requirement only:

```
$ composer require --dev zendframework/zend-component-installer
```

(Note: it will likely already have been installed with zend-expressive-tooling; requiring it like this, however, ensures it stays present if you decide to remove that package later.)

To update your application, you will need to update your `config/config.php` file.

If you've made no modifications to the shipped version, it will look like the following:

```php
<?php

use Zend\Stdlib\ArrayUtils;
use Zend\Stdlib\Glob;

/**
 * Configuration files are loaded in a specific order. First ``global.php``, then ``*.
global.php``.
 * then ``local.php`` and finally ``*.local.php``. This way local settings overwrite g
lobal settings.
 *
 * The configuration can be cached. This can be done by setting ``config_cache_enabled
`` to ``true``.
 *
 * Obviously, if you use closures in your config you can't cache it.
 */

$cachedConfigFile = 'data/cache/app_config.php';

$config = [];
if (is_file($cachedConfigFile)) {
    // Try to load the cached config
    $config = include $cachedConfigFile;
} else {
    // Load configuration from autoload path
    foreach (Glob::glob('config/autoload/{{,*.}global,{,*.}local}.php', Glob::GLOB_BRA
CE) as $file) {
        $config = ArrayUtils::merge($config, include $file);
    }

    // Cache config if enabled
    if (isset($config['config_cache_enabled']) && $config['config_cache_enabled'] ===
true) {
        file_put_contents($cachedConfigFile, '<?php return ' . var_export($config, true
) . ';');
    }
}

// Return an ArrayObject so we can inject the config as a service in Aura.Di
// and still use array checks like ``is_array``.
return new ArrayObject($config, ArrayObject::ARRAY_AS_PROPS);
```

You can replace it directly with this, then:

```php
<?php

use Zend\ConfigAggregator\ArrayProvider;
use Zend\ConfigAggregator\ConfigAggregator;
use Zend\ConfigAggregator\PhpFileProvider;

$cacheConfig = [
    'config_cache_path' => 'data/config-cache.php',
];

$aggregator = new ConfigAggregator([
    new ArrayProvider($cacheConfig),

    new PhpFileProvider('config/autoload/{{,*.}global,{,*.}local}.php'),
], $cacheConfig['config_cache_path']);

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

If you want, you can set the `config_cache_path` to match the one from your previous version; this should only be necessary if you have tooling already in place for cache clearing, however.

> ### ZF components
>
> Any Zend Framework component that provides service configuration exposes a `ConfigProvider`. This means that if you add these to your application after making the above changes, they will expose their services to your application immediately following installation!
>
> If you've installed ZF components prior to this change, check to see which ones expose `ConfigProvider` classes (you can look for a `ConfigProvider` under their namespace, or look for an `extra.zf.config-provider` declaration in their `composer.json`). If you find any, add them to your `config/config.php` file; using the fully qualified class name of the provider. As an example: `\Zend\Db\ConfigProvider::class`.

Development mode

We have been using zf-development-mode with zend-mvc and Apigility applications for a few years now, and feel it offers an elegant solution for shipping standard development configuration for use with your team, as well as toggling back and forth between development and production configuration. (That said, `config/autoload/*.local.php` files may clearly vary in your development environment versus your production environment, so this is not entirely fool-proof!)

Let's add it to our application:

```
$ composer require --dev zfcampus/zf-development-mode
```

Note that we're adding it as a development requirement; chances are, you do not want to accidentally enable it in production!

Next, we need to add a couple files to our tree. The first we'll add is `config/development.config.php.dist`, with the following contents:

```php
<?php

/**
 * File required to allow enablement of development mode.
 *
 * For use with the zf-development-mode tool.
 *
 * Usage:
 *   $ composer development-disable
 *   $ composer development-enable
 *   $ composer development-status
 *
 * DO NOT MODIFY THIS FILE.
 *
 * Provide your own development-mode settings by editing the file
 * `config/autoload/development.local.php.dist`.
 *
 * Because this file is aggregated last, it simply ensures:
 *
 * - The `debug` flag is _enabled_.
 * - Configuration caching is _disabled_.
 */

use Zend\ConfigAggregator\ConfigAggregator;

return [
    'debug' => true,
    ConfigAggregator::ENABLE_CACHE => false,
];
```

Next, we'll add a `config/autoload/development.local.php.dist`. The contents of this one will vary based on what you are using in your application.

If you are **not** using Whoops for error reporting, start with this:

```php
<?php
return [
];
```

If you are, this is a chance to configure that correctly for your newly updated application. Create the file with these contents:

```
<?php

use Whoops\Handler\PrettyPageHandler;
use Zend\Expressive\Container;
use Zend\Expressive\Middleware\ErrorResponseGenerator;
use Zend\Expressive\Whoops;
use Zend\Expressive\WhoopsPageHandler;

return [
    'dependencies' => [
        'invokables' => [
            WhoopsPageHandler::class => PrettyPageHandler::class,
        ],
        'factories' => [
            ErrorResponseGenerator::class => Container\WhoopsErrorResponseGeneratorFac
tory::class,
            Whoops::class => Container\WhoopsFactory::class,
        ],
    ],

    'whoops' => [
        'json_exceptions' => [
            'display'    => true,
            'show_trace' => true,
            'ajax_only'  => true,
        ],
    ],
];
```

Next, if you started with the V1 skeleton application, you will likely have a file named `config/autoload/errorhandler.local.php`, and it will have similar contents, for the purpose of seeding the legacy "final handler" system. You can now *remove* that file.

After that's done, we need to add some directives so that git will ignore the non-dist files. Edit the `.gitignore` file in your project's root directory to add the following entry:

```
config/development.config.php
```

The `config/autoload/.gitignore` file should already have a rule that omits `*.local.php`.

Now we need to have our configuration load the development configuration if it's present. The following assumes you already converted your application to use zend-config-aggregator. Add the following line as the last element of the array passed when instantiating your `ConfigAggregator`:

```
    new PhpFileProvider('config/development.config.php'),
```

If the file is missing, that provider will return an empty array; if it's present, it returns whatever configuration the file returns. By making it the last element merged, we can do things like override configuration caching, and force debug mode, which is what our `config/development.config.php.dist` file does!

Finally, let's add some convenience scripts to composer. Open your `composer.json` file, find the `scripts` section, and add the following to it:

```
        "development-disable": "zf-development-mode disable",
        "development-enable": "zf-development-mode enable",
        "development-status": "zf-development-mode status",
```

Now we can try it out!

Run:

```
$ composer development-status
```

This should tell you that development mode is currently disabled.

Next, run:

```
$ composer development-enable
```

This will enable development mode.

If you want to test and ensure you're in development mode, edit one of your middleware to have it raise an exception, and see what happens!

Clean up

If your application is working correctly, you can now do some additional cleanup.

- Edit your `config/autoload/middleware-pipeline.global.php` file to remove the `middleware_pipeline` key and its contents.
- Edit your `config/autoload/routes.global.php` file to remove the `routes` key and its contents.
- Search for any references to a `FinalHandler` within your dependency configuration, and remove them.

At this point, you should have a fully working Expressive 2 application!

Final step: Updating your middleware

Now that the initial migration is complete, you can take some more steps!

One of the big changes is that Expressive 2 *prefers* middleware implementing http-interop/http-middleware's `MiddlewareInterface`. This requires a few changes to your middleware.

First, let's look at the interfaces defined by http-interop/http-middleware:

```
namespace Interop\Http\ServerMiddleware;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface MiddlewareInterface
{
    /**
     * @return ResponseInterface
     */
    public function process(ServerRequestInterface $request, DelegateInterface $delega
te);
}

interface DelegateInterface
{
    /**
     * @return ResponseInterface
     */
    public function process(ServerRequestInterface $request);
}
```

The first interface defines middleware. Unlike Expressive 1, http-interop middleware does not receive a response instance. There are a variety of reasons for this, but Anthony Ferrara sums them up best in a blog post he wrote in May 2016[4].

Another difference is that instead of a `callable $next` argument, we have a `DelegateInterface $delegate`. This provides better type-safety, and, because each of the `MiddlewareInterface` and `DelegateInterface` define the same `process()` method, ensures that implementations of middleware and delegates are discrete and do not mix concerns. *Delegates* are classes that can process a request if the *current middleware* cannot fully do so. Examples might include middleware that will inject additional response headers, or middleware that only acts when certain request criteria are present (such as HTTP caching headers).

The upshot is that when rewriting your middleware to use the new interfaces, you need to do several things:

- First, import the http-interop interfaces into your class file:

  ```
  use Interop\Http\ServerMiddleware\DelegateInterface;
  use Interop\Http\ServerMiddleware\MiddlewareInterface;
  ```

- Second, rename the `__invoke()` method to `process()`.
- Third, update the signature of your new `process` method to be:

  ```
  public function process(ServerRequestInterface $request, DelegateInterface $delega
  te)
  ```

- Fourth, look for calls to `$next()`. As an example, the following:

  ```
  return $next($request, $response);
  ```

  Becomes:

  ```
  return $delegate->process($request);
  ```

  These updates will vary on a case-by-case basis: in some cases, you may be calling methods on the request instance; in other cases, you may be capturing the returned response

- Look for cases where you were using the passed `$response` instance, and eliminate those. You may do so as follows:
  - Use the response returned by calling `$delegate->process()` instead.
  - Create a new concrete response instance and operate on it.
  - Compose a "response prototype" in your middleware if you do not want to create a new response instance directly, and operate on it. Doing so will require that you update any factory associated with the middleware class, however.

As an example, let's look at a simple middleware that adds a response header:

```
namespace App\Middleware;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

class TheClacksMiddleware
{
    public function __invoke(ServerRequestInterface $request, ResponseInterface $respo
nse, callable $next)
    {
        $response = $next($request, $response);

        return $response->withHeader('X-Clacks-Overhead', ['GNU Terry Pratchett']);
    }
}
```

When we refactor it to be http-interop middleware, it becomes:

```
namespace App\Middleware;

use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface;
use Psr\Http\Message\ServerRequestInterface;

class TheClacksMiddleware implements MiddlewareInterface
{
    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        $response = $delegate->process($request);

        return $response->withHeader('X-Clacks-Overhead', ['GNU Terry Pratchett']);
    }
}
```

Summary

Migration consists of:

- Updating dependencies.
- Running migration scripts provided by zendframework/zend-expressive-tooling.
- Optionally adding a self-invoking function around code creating variables in `public/index.php` .
- Optionally updating your application to use zendframework/zend-config-aggregator for configuration aggregation.
- Optionally adding zfcampus/zf-development-mode integration to your application.

- Optionally updating your middleware to implement http-interop/http-middleware.

As noted, many of these changes are optional. Your application will continue to run without them. Updating them will modernize your application, however, and make it more familiar to developers familiar with the Expressive 2 skeleton.

We hope this guide gets you successfully migrated! If you run into issues not covered here, please let us know via an issue on the Expressive repository[5].

Footnotes

> 1. https://framework.zend.com/blog/2017-03-07-expressive-2.html ↩

> 2. https://github.com/adamculp/expressive-blastoff ↩

> 3. https://docs.zendframework.com/zend-expressive/features/modular-applications/ ↩

> 4. http://blog.ircmaxell.com/2016/05/all-about-middleware.html ↩

> 5. https://github.com/zendframework/zend-expressive/issues/new ↩

Develop Expressive Applications Rapidly Using CLI Tooling

by Matthew Weier O'Phinney

First impressions matter, particularly when you start using a new framework. As such, we're striving to improve your first tasks with Expressive.

With the 2.0 release, we provided several migration tools, as well as tooling for creating, registering, and deregistering middleware modules. Each was shipped as a separate script, with little unification between them.

Today, we've pushed a unified script, `expressive` , which provides access to all the migration tooling, module tooling, and new tooling to help you create http-interop middleware. Our hope is to make your first few minutes with Expressive a bit easier, so you can start writing powerful applications.

Getting the tooling

If you haven't created an application yet:

```
$ composer create-project zendframework/zend-expressive-skeleton
```

will create a new project using the latest 2.0.2 release, which contains the new `expressive` script.

If you are already using Expressive 2, you can get the latest tooling using the following, regardless of whether or not you've previously installed it:

```
$ composer require --dev "zendframework/zend-expressive-tooling:^0.4.1"
```

What tooling do you get?

The `expressive` script has three general categories of commands:

- **`migrate:*`** : these are intended for Expressive 1 users who are migrating to Expressive 2. We'll ignore these for now, as we covered them in the previous chapter.
- **`module:*`** Create, register, and deregister Expressive middleware modules.

- `middleware:*` : Create http-interop middleware class files.

Create your first module

For purposes of illustration, we'll consider that you want to create an API for listing books. You anticipate that the functionality can be self-contained, and that you may want to potentially extract it later to re-use elsewhere. As such, you have a good case for creating a module[1].

Let's get started:

```
$ ./vendor/bin/expressive module:create BooksApi
```

The above does the following:

- It creates a directory tree for a `BooksApi` module under `src/BooksApi/`, with a subtree for source code, and another for templates.
- It creates the class `BooksApi\ConfigProvider` in the file `src/BooksApi/src/ConfigProvider.php`
- It adds a PSR-4 autoloader entry for `BooksApi` in your `composer.json`, and runs `composer dump-autoload` to ensure the new autoloader rule is generated within your application.
- It adds an entry for the generated `BooksApi\ConfigProvider` to your `config/config.php` file.

At this point, we have a module with no code! Let's rectify that situation!

Create middleware

We know we will want to *list books*, so we'll create middleware for that:

```
$ ./vendor/bin/expressive middleware:create "BooksApi\Action\ListBooksAction"
```

> ### Use quotes!
>
> PHP's namespace separator is the backslash, which is typically interpreted as an escape character in most shells. As such, use double or single quotes around the middleware name to ensure it is passed correctly to the command!

This creates the class `BooksApi\Action\ListBooksAction` in the file `src/BooksApi/src/Action/ListBooksAction.php`. In doing so, it creates the `src/BooksApi/src/Action/` directory, as it did not previously exist!

The class file contents will look like this:

```php
namespace BooksApi\Action;

use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface;
use Psr\Http\ServerRequestInterface;

class ListBooksAction implements MiddlewareInterface
{
    /**
     * {@inheritDoc}
     */
    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        // $response = $delegate->process($request);
    }
}
```

At this point, you're ready to start coding!

Future direction

This tooling is just a start; we're well aware that developers will want and need more tooling to make development more convenient. As such, we have a call to action: please open issues[2] to request more commands that will make your life easier, or open pull requests that implement the tools you need. If you are unsure how to do so, use the existing code to get an idea of how to proceed, or ask in the #expressive-contrib Slack channel[3].

Footnotes

1. https://docs.zendframework.com/zend-expressive/features/modular-applications/ ↩
2. https://github.com/zendframework/zend-expressive-tooling/issues/new ↩
3. Get an invite to our Slack at https://zendframework-slack.herokuapp.com ↩

Nested Middleware in Expressive

by Matthew Weier O'Phinney

A major reason to adopt a middleware architecture is the ability to create custom workflows for your application. Most traditional MVC architectures have a very specific workflow the request follows. While this is often customizable via event listeners, the events and general request lifecycle is the same for each and every resource the application serves.

With middleware, however, you can define your own workflow by composing middleware.

Expressive pipelines

In Expressive, we call the workflow the application *pipeline*, and you create it by *piping* middleware into the application. As an example, the default pipeline installed with the skeleton application looks like this:

```
// In config/pipeline.php:
use Zend\Expressive\Helper\ServerUrlMiddleware;
use Zend\Expressive\Helper\UrlHelperMiddleware;
use Zend\Expressive\Middleware\ImplicitHeadMiddleware;
use Zend\Expressive\Middleware\ImplicitOptionsMiddleware;
use Zend\Expressive\Middleware\NotFoundHandler;
use Zend\Stratigility\Middleware\ErrorHandler;

$app->pipe(ErrorHandler::class);
$app->pipe(ServerUrlMiddleware::class);
$app->pipeRoutingMiddleware();
$app->pipe(ImplicitHeadMiddleware::class);
$app->pipe(ImplicitOptionsMiddleware::class);
$app->pipe(UrlHelperMiddleware::class);
$app->pipeDispatchMiddleware();
$app->pipe(NotFoundHandler::class);
```

In this particular workflow, what happens when a request is processed is the following:

- The `ErrorHandler` middleware (which handles exceptions and PHP errors) is processed, which in turn:
    - processes the `ServerUrlMiddleware` (which injects the request URI into the `ServerUrl` helper), which in turn:
        - process the routing middleware, which in turn:
            - process the `ImplicitHeadMiddleware` (which provides responses for `HEAD`

requests if the matched middleware does not handle that method), which in turn:

- processes the `ImplicitOptionsMiddleware` (which provides responses for `OPTIONS` requests if the matched middleware does not handle that method), which in turn:
  - processes the `UrlHelperMiddleware` (which injects the `UrlHelper` with the `RouteResult` from routing, if discovered), which in turn:
    - processes the dispatch middleware, which in turn:
      - processes the matched middleware, if present
      - processes the `NotFoundHandler`, if no middleware was matched by routing, or that middleware cannot handle the request.

At any point in the workflow, middleware can choose to return a response. For instance, the `ImplicitHeadMiddleware` and `ImplicitOptionsMiddleware` may return a response if the middleware matched by routing cannot handle the specified method. When they do, no layers below are executed!

Scenario: Adding Authentication

Now, let's say we want to add authentication to our application.

For purposes of this example, we'll use the `BasicAuthentication` middleware[1] from the middlewares/http-authentication package[2]:

```
$ composer require middlewares/http-authentication
```

When this middleware executes, it looks at the various HTTP request headers used for HTTP Basic Authentication, and then attempts to verify the credentials against a list composed in the instance. If login fails, the middleware returns a 401 response; otherwise, it delegates to the next middleware.

The middleware accepts a list of username/password pairs to its constructor. It also allows you to provide an authentication realm via the `realm()` method, and the attribute to which to save the name of the authenticated user within the request used to dispatch the next middleware when authentication succeeds. We'll create a factory to configure the middleware:

```php
<?php
namespace Acme;

use Middlewares\BasicAuthentication
use Psr\Container\ContainerInterface;

class BasicAuthenticationFactory
{
    /**
     * @return BasicAuthentication
     */
    public function __invoke(ContainerInterface $container)
    {
        $config = $container->has('config') ? $container->get('config') : [];
        $credentials = $config['authentication']['credentials'] ?? [];
        $realm = $config['authentication']['realm'] ?? __NAMESPACE__;
        $attribute = $config['authentication']['attribute']
            ??  BasicAuthentication::class;

        $middleware = new BasicAuthentication($credentials);
        $middleware->realm($realm);
        $middleware->attribute($attribute);

        return $middleware;
    }
}
```

Wire this in your dependencies somewhere; we recommend either the file `config/autoload/dependencies.global.php` or the class `Acme\ConfigProvider` if you have defined it:

```php
'dependencies' => [
    'factories' => [
        Middlewares\BasicAuthentication::class => Acme\BasicAuthenticationFactory::cla
ss,
    ],
],
```

Now, we'll add this to the pipeline.

If you want *every* request to require authentication, you can pipe this in early, sometime after the `ErrorHandler` and any middleware you want to run for every request:

```php
// In config/pipeline.php:

$app->pipe(ErrorHandler::class);
$app->pipe(ServerUrlMiddleware::class);
$app->pipe(\Middlewares\BasicAuthentication::class);
```

Done!

But... this means that *all* pages of the application now require authentication! You likely don't want to require authentication for the home page, and potentially many others.

Let's look at some options.

Segregating by path

One option available in Expressive is *path segregation*. If you know every route requiring authentication will have the same path prefix, you can use this approach.

As an example, let's say you only want authentication for your API, and all API paths fall under the path prefix `/api`. This means you could do the following:

```
$app->pipe('/api', \Middlewares\BasicAuthentication::class);
```

This middleware will only execute if the request path matches `/api`.

But what if you only really need authentication for *specific routes* under the API?

Nested middleware

We finally get to the purpose of this tutorial!

Let's say our API defines the following routes:

```
// In config/routes.php:

$app->get('/api/books', Acme\Api\BookListMiddleware::class, 'api.books');
$app->post('/api/books', Acme\Api\CreateBookMiddleware::class);
$app->get('/api/books/{book_id:\d+}', Acme\Api\BookMiddleware::class, 'api.book');
$app->patch('/api/books/{book_id:\d+}', Acme\Api\UpdateBookMiddleware::class);
$app->delete('/api/books/{book_id:\d+}', Acme\Api\DeleteBookMiddleware::class);
```

In this scenario, we want to require authentication only for the `CreateBookMiddleware`, `UpdateBookMiddleware`, and `DeleteBookMiddleware`. How do we do that?

Expressive allows you to provide a *list* of middleware both when piping and routing, instead of a single middleware. Just as when you specify a single middleware, each entry may be one of:

- callable middleware

- middleware instance
- service name resolving to middleware

Internally, Expressive creates a `Zend\Stratigility\MiddlewarePipe` instance with the specified middleware, and processes this pipeline when the given middleware is matched.

So, going back to our previous example, where we defined routes, we can rewrite them as follows:

```php
// In config/routes.php:

$app->get('/api/books', Acme\Api\BookListMiddleware::class, 'api.books');
$app->post('/api/books', [
    Middlewares\BasicAuthentication::class,
    Acme\Api\CreateBookMiddleware::class,
]);
$app->get('/api/books/{book_id:\d+}', Acme\Api\BookMiddleware::class, 'api.book');
$app->patch('/api/books/{book_id:\d+}', [
    Middlewares\BasicAuthentication::class,
    Acme\Api\UpdateBookMiddleware::class,
]);
$app->delete('/api/books/{book_id:\d+}', [
    Middlewares\BasicAuthentication::class,
    Acme\Api\DeleteBookMiddleware::class,
]);
```

In this particular case, this means that the `BasicAuthentication` middleware will only execute for one of the following:

- `POST` requests to `/api/books`
- `PATCH` requests to `/api/books/123` (or any valid identifier)
- `DELETE` requests to `/api/books/123` (or any valid identifier)

In each case, if authentication fails, the later middleware in the list *will not be processed*, as the `BasicAuthentication` middleware will return a 401 response.

This technique allows for some powerful workflows. For instance, when creating a book via the `/api/books` middleware, we could also add in middleware to check the content type, parse the incoming request, and validate the submitted data:

```php
// In config/routes.php:

$app->post('/api/books', [
    Middlewares\BasicAuthentication::class,
    Acme\ContentNegotiationMiddleware::class,
    Zend\Expressive\Helper\BodyParams\BodyParamsMiddleware::class,
    Acme\Api\BookValidationMiddleware::class,
    Acme\Api\CreateBookMiddleware::class,
]);
```

(We leave implementation of most of the above middleware as an exercise for the reader!)

By using service names, you also ensure that optimal performance; the middleware will not be instantiated unless the request matches, and the middleware is executed. In fact, if one of the pipeline middleware for the given route returns a response early, even the middleware later in the queue will not be instantiated!

> ### A note about order
>
> When you create middleware pipelines such as the above, as well as in the following examples, *order matters*. Pipelines are managed internally as queues, and thus are first-in-first-out (FIFO). As such, putting the responding `CreateBookMiddleware` (which will most likely return a response with the API payload) will result in the other middleware never executing!
>
> As such, ensure that your pipelines contain middleware that will *delegate* first, and your primary middleware that returns a response last.

Middleware pipelines

Another approach would be to setup a middleware pipeline manually within the factory for the requested middleware. The following examples creates and returns a `Zend\Stratigility\MiddlewarePipe` instance that composes the same middleware as in the previous example that used a list of middleware when routing, returning the `MiddlewarePipe` instead of the requested `CreateBookMiddleware` (but composing it nonetheless):

```
namespace Acme\Api;

use Acme\ContentNegotiationMiddleware;
use Middlewares\BasicAuthentication;
use Psr\Container\ContainerInterface;
use Zend\Expressive\Helper\BodyParams\BodyParamsMiddleware;
use Zend\Stratigility\MiddlewarePipe;

class CreateBookMiddlewareFactory
{
    public function __invoke(ContainerInterface $container)
    {
        $pipeline = new MiddlewarePipe();

        $pipeline->pipe($container->get(BasicAuthentication::class));
        $pipeline->pipe($container->get(ContentValidationMiddleware::class));
        $pipeline->pipe($container->get(BodyParamsMiddleware::class));
        $pipeline->pipe($container->get(BookValidationMiddleware::class));

        // If dependencies are needed, pull them from the container and pass
        // them to the constructor:
        $nested->pipe(new CreateBookMiddleware());

        return $pipeline;
    }
}
```

This approach is inferior to using an array of middleware, however. Internally, Expressive will wrap the various middleware services you list in `LazyLoadingMiddleware` instances; this means that if a service earlier in the pipeline returns early, the service will never be pulled from the container. This can be important if any services might establish network connections or perform file operations during initialization!

Nested applications

Since Expressive does the work of lazy loading services, another option would be to create another Expressive `Application` instance, and feed it, instead of creating a `MiddlewarePipe`:

```php
namespace Acme\Api;

use Acme\ContentNegotiationMiddleware;
use Middlewares\BasicAuthentication;
use Psr\Container\ContainerInterface;
use Zend\Expressive\Application;
use Zend\Expressive\Helper\BodyParams\BodyParamsMiddleware;
use Zend\Expressive\Router\RouterInterface;

class CreateBookMiddlewareFactory
{
    public function __invoke(ContainerInterface $container)
    {
        $nested = new Application(
          $container->get(RouterInterface::class),
          $container
        );

        $nested->pipe(BasicAuthentication::class);
        $nested->pipe(ContentValidationMiddleware::class);
        $nested->pipe(BodyParamsMiddleware::class);
        $nested->pipe(BookValidationMiddleware::class);

        // If dependencies are needed, pull them from the container and pass
        // them to the constructor:
        $nested->pipe(new CreateBookMiddleware());

        return $nested;
    }
}
```

The benefit this approach has is that you get the lazy-loading middleware instances without effort. However, it makes discovery of what the middleware consists more difficult — you can't just look at the routes anymore, but need to look at the factory itself to see what the workflow looks like. When you consider re-distribution and re-use, though, this approach has a lot to offer, as it combines the performance of defining an application pipeline with the ability to re-use that same workflow any time you use that particular middleware in an application.

(The above could even use separate router and container instances entirely, in order to keep the services and routing for the middleware pipeline completely separate from those of the main application!)

Using traits for common workflows

The above approach of creating a nested application, as well as the original example of nested middleware provided via arrays, has one drawback: if several middleware need the exact same workflow, you'll have repetition.

One approach is to create a trait[3] for creating the `Application` instance and populating the initial pipeline.

```
namespace Acme\Api;

use Acme\ContentNegotiationMiddleware;
use Middlewares\BasicAuthentication;
use Psr\Container\ContainerInterface;
use Zend\Expressive\Application;
use Zend\Expressive\Helper\BodyParams\BodyParamsMiddleware;
use Zend\Expressive\Router\RouterInterface;

trait CommonApiPipelineTrait
{
    private function createNestedApplication(ContainerInterface $container)
    {
        $nested = new Application(
          $container->get(RouterInterface::class),
          $container
        );

        $nested->pipe(BasicAuthentication::class);
        $nested->pipe(ContentValidationMiddleware::class);
        $nested->pipe(BodyParamsMiddleware::class);
        $nested->pipe(BookValidationMiddleware::class);

        return $nested;
    }
}
```

Our `CreateBookMiddlewareFactory` then becomes:

```
namespace Acme\Api;

use Psr\Container\ContainerInterface;

class CreateBookMiddlewareFactory
{
    use CommonApiPipelineTrait;

    public function __invoke(ContainerInterface $container)
    {
        $nested = $this->createNestedApplication($container);

        // If dependencies are needed, pull them from the container and pass
        // them to the constructor:
        $nested->pipe(new CreateBookMiddleware());

        return $nested;
    }
}
```

Any middleware that would need the same workflow can now provide a factory that uses the same trait. This, of course, means that the factories for any given middleware that adopts the specific workflow reflect that, meaning they cannot e re-used without using that specific workflow.

Delegator factories

To solve this latter problem — allowing re-use of middleware without requiring the specific pipeline — we provide another approach: delegator factories[4].

Available since version 2 of the Expressive skeleton, delegator factories intercept creation of a service, and allow you to act on the service before returning it, or replace it with another instance entirely!

The above trait could be rewritten as a delegator factory:

```php
namespace Acme\Api;

use Middlewares\BasicAuthentication;
use Acme\ContentNegotiationMiddleware;
use Psr\Container\ContainerInterface;
use Zend\Expressive\Application;
use Zend\Expressive\Helper\BodyParams\BodyParamsMiddleware;
use Zend\Expressive\Router\RouterInterface;

class CommonApiPipelineDelegatorFactory
{
    public function __invoke(ContainerInterface $container, $name, callable $callback)
    {
        $nested = new Application(
          $container->get(RouterInterface::class),
          $container
        );

        $nested->pipe(BasicAuthentication::class);
        $nested->pipe(ContentValidationMiddleware::class);
        $nested->pipe(BodyParamsMiddleware::class);
        $nested->pipe(BookValidationMiddleware::class);

        // Inject the middleware service requested:
        $nested->pipe($callback());

        return $nested;
    }
}
```

You could then register this with any service that needs the pipeline, without needing to change their factories. As an example, you could have the following in either the `config/autoload/dependencies.global.php` file or the `Acme\ConfigProvider` class, if defined:

```
'dependencies' => [
    'factories' => [
        \Acme\Api\CreateBookMiddleware::class => \Acme\Api\CreateBookMiddlewareFactory
::class,
        \Acme\Api\DeleteBookMiddleware::class => \Acme\Api\DeleteBookMiddlewareFactory
::class,
        \Acme\Api\UpdateBookMiddleware::class => \Acme\Api\UpdateBookMiddlewareFactory
::class,
    ],
    'delegators' => [
        \Acme\Api\CreateBookMiddleware::class => [
            \Acme\Api\CommonApiPipelineDelegatorFactory::class,
        ],
        \Acme\Api\DeleteBookMiddleware::class => [
            \Acme\Api\CommonApiPipelineDelegatorFactory::class,
        ],
        \Acme\Api\UpdateBookMiddleware::class => [
            \Acme\Api\CommonApiPipelineDelegatorFactory::class,
        ],
    ],
],
```

This approach offers re-usability even when a given middleware may not have expected to be used in a specific workflow!

Middleware all the way down!

We hope this tutorial demonstrates the power and flexibility of Expressive, and how you can create workflows that are granular even to specific middleware. We covered a number of features in this post:

- Pipeline middleware that operates for all requests.
- Path-segregated middleware.
- Middleware nesting via lists of middleware.
- Returning pipelines or applications from individual service factories.
- Using delegator factories to create and return nested pipelines or applications.

Footnotes

> 1. https://github.com/middlewares/http-authentication#basicauthentication ↩

> 2. https://github.com/middlewares/http-authentication ↩

> 3. http://php.net/trait ↩

4

> 4. https://docs.zendframework.com/zend-expressive/features/container/delegator-factories/ ↩

Error Handling in Expressive

by Matthew Weier O'Phinney

One of the big improvements in Expressive 2 is how error handling is approached. While the error handling documentation[1] covers the feature in detail, more examples are never a bad thing!

Our scenario

For our example, we'll create an API resource that returns a list of books read. Being an API, we want to return JSON; this is true even when we want to present error details. Our challenge, then, will be to add error handling that presents JSON error details when the API is invoked — but use the existing error handling otherwise.

The middleware

The middleware looks like the following:

```php
// In src/Acme/BooksRead/ListBooksRead.php:

namespace Acme\BooksRead;

use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface;
use PDO;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\JsonResponse;

class ListBooksRead implements MiddlewareInterface
{
    const SORT_ALLOWED = [
        'author',
        'date',
        'title',
    ];

    const SORT_DIR_ALLOWED = [
        'ASC',
        'DESC',
    ];

    private $pdo;
    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        $query   = $request->getQueryParams();
        $page    = $this->validatePageOrPerPage((int) ($query['page'] ?? 1));
        $perPage = $this->validatePageOrPerPage((int) ($query['per_page'] ?? 25));
        $sort    = $this->validateSort($query['sort'] ?? 'date');
        $sortDir = $this->validateSortDirection($query['sort_direction'] ?? 'DESC');

        $offset = ($page - 1) * $perPage;

        $statement = $pdo->prepare(sprintf(
            'SELECT * FROM books_read ORDER BY %s %s LIMIT %d OFFSET %d',
            $sort,
            $sortDir,
            $perPage,
            $offset
        ));

        try {
            $statement->execute([]);
        } catch (PDOException $e) {
            throw Exception\ServerError::create(
                'Database error occurred',
                sprintf('A database error occurred: %s', $e->getMessage()),
                ['trace' => $e->getTrace()]
            );
        }

        $books = $statement->fetchAll(PDO::FETCH_ASSOC);

        return new JsonResponse(['books' => $books]);
    }

    private function validatePageOrPerPage($value, $param)
    {
        if ($value > 1) {
            return $value;
        }

        throw Exception\InvalidRequest::create(
            sprintf('Invalid %s value specified', $param),
            sprintf('The %s specified must be an integer greater than 1', $param)
        );
    }

    private function validateSort(string $sort)
    {
        if (in_array($sort, self::SORT_ALLOWED, true)) {
            return $sort;
        }

        throw Exception\InvalidRequest::create(
            'Invalid sort type specified',
            sprintf(
                'The sort type specified must be one of [ %s ]',
                implode(', ', self::SORT_ALLOWED)
            )
        );
    }

    private function validateSortDirection(string $direction)
    {
        if (in_array($direction, self::SORT_DIR_ALLOWED, true)) {
            return $direction;
        }

        throw Exception\InvalidRequest::create(
            'Invalid sort direction specified',
            sprintf(
                'The sort direction specified must be one of [ %s ]',
                implode(', ', self::SORT_DIR_ALLOWED)
            )
        );
    }
}
```

You'll notice that this middleware throws exceptions for error handling, and uses some custom exception types. Let's examine those next.

The exceptions

Our API will have custom exceptions. In order to provide *useful* details to our users, we'll have our exceptions compose additional details that we can report. As such, we'll have a special interface for our API exceptions that exposes the custom details.

We'll also define a few specific types. Since much of the work will be the same between these types, we'll use a trait to define the common code, and compose that into each.

```
// In src/Acme/BooksRead/Exception/MiddlewareException.php:

namespace Acme\BooksRead\Exception;

interface MiddlewareException
{
    public static function create() : MiddlewareException;
    public function getStatusCode() : int;
    public function getType() : string;
    public function getTitle() : string;
    public function getDescription() : string;
    public function getAdditionalData() : array;
}
```

```
// In src/Acme/BooksRead/Exception/MiddlewareExceptionTrait.php:

namespace Acme\BooksRead\Exception;

trait MiddlewareExceptionTrait
{
    private $statusCode;
    private $title;
    private $description;
    private $additionalData = [];

    public function getStatusCode() : int
    {
        return $this->statusCode;
    }

    public function getTitle() : string
    {
        return $this->title;
    }

    public function getDescription() : string
    {
        return $this->description;
    }

    public function getAdditionalData() : array
    {
        return $this->additionalData;
    }
}
```

```php
// In src/Acme/BooksRead/Exception/ServerError.php:

namespace Acme\BooksRead\Exception;

use RuntimeException;

class ServerError extends RuntimeException implements MiddlewareException
{
    use MiddlewareExceptionTrait;

    public static function create(string $title, string $description, array $additiona
lData = [])
    {
        $e = new self($description, 500);
        $e->statusCode = 500;
        $e->title = $title;
        $e->additionalData = $additionalData;
        return $e;
    }

    public function getType() : string
    {
        return 'https://example.com/api/problems/server-error';
    }
}
```

```
// In src/Acme/BooksRead/Exception/InvalidRequest.php:

namespace Acme\BooksRead\Exception;

use RuntimeException;

class InvalidRequest extends RuntimeException implements MiddlewareException
{
    use MiddlewareExceptionTrait;

    public static function create(string $title, string $description, array $additiona
lData = [])
    {
        $e = new self($description, 400);
        $e->statusCode = 400;
        $e->title = $title;
        $e->additionalData = $additionalData;
        return $e;
    }

    public function getType() : string
    {
        return 'https://example.com/api/problems/invalid-request';
    }
}
```

These specialized exception types have additional methods for retrieving additional data. Furthermore, they set default exception codes, which may be repurposed as status codes.

A Problem Details error handler

What we want to have happen is for our API to return data in Problem Details[2] format.

To accomplish this, we'll create a new middleware that will catch our domain-specific exception type in order to create an appropriate response for us.

```
// In src/Acme/BooksRead/ProblemDetailsMiddleware.php:

namespace Acme\BooksRead;

use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface;
use Psr\Http\Message\ServerRequestInterface;
use Throwable;
use Zend\Diactoros\Response\JsonResponse;

class ProblemDetailsMiddleware implements MiddlewareInterface
{
    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        try {
            $response = $delegate->process($request);
            return $response;
        } catch (Exception\MiddlewareException $e) {
            // caught; we'll handle it following the try/catch block
        } catch (Throwable $e) {
            throw $e;
        }

        $problem = [
            'type'   => $e->getType(),
            'title'  => $e->getTitle(),
            'detail' => $e->getDescription(),
        ];
        $problem = array_merge($e->getAdditionalData(), $problem);

        return new JsonResponse($problem, $e->getStatusCode(), [
            'Content-Type' => 'application/problem+json',
        ]);
    }
}
```

This middleware always delegates processing of the request, but does so in a try/catch block. If it catches our special `MiddlewareException`, it will process it; otherwise, it re-throws the caught exception, to allow middleware in an outer layer to handle it.

Composing the error handler

> In our previous article, Nested Middleware in Expressive, we detail how to nest middleware pipelines and create nested middleware pipelines for routed middleware. We'll use those techniques here. Please read that before continuing.

Assuming we have already defined a factory for our `ListBooksRead` middleware (likely class `Acme\BooksRead\ListBooksReadFactory`, in `src/Acme/BooksRead/ListBooksReadFactory.php`), we have a few options. First, we could compose this error handler in a middleware pipeline within our routing configuration:

```
// In config/routes.php:

$app->get('/api/books-read', [
    \Acme\BooksRead\ProblemDetailsMiddleware::class,
    \Acme\BooksRead\ListBooksRead::class,
], 'api.books-read')
```

If there are other concerns — such as authentication, authorization, content negotiation, etc. — you may want to instead create a delegator factory; this can then be re-used for other API resources that need the same set of middleware. As an example:

```
// In src/Acme/BooksRead/ApiMiddlewareDelegatorFactory.php:

namespace Acme\BooksRead;

use Psr\Container\ContainerInterface;
use Zend\Expressive\Application;
use Zend\Expressive\Router\RouterInterface;

class ApiMiddlewareDelegatorFactory
{
    public function __invoke(ContainerInterface $container, $name, callable $callback)
    {
        $apiPipeline = new Application(
            $container->get(RouterInterface::class),
            $container
        );

        $apiPipeline->pipe(ProblemDetailsMiddleware::class);
        // ..and pipe other middleware as necessary...

        $apiPipeline->pipe($callback());

        return $apiPipeline;
    }
}
```

The above would then be registered as a delegator with your `ListBooksRead` service:

```
// In Acme\BooksRead\ConfigProvider, or any config/autoload/*.global.php:

return [
    'dependencies' => [
        'delegators' => [
            \Acme\BooksRead\ListBooksRead::class => [
                \Acme\BooksRead\ApiMiddlewareDelegatorFactory::class,
            ],
        ],
    ]
];
```

End result

Once you have created the pipeline, you should get some nice errors:

```
HTTP/1.1 400 Client Error
Content-Type: application/problem+json

{
  "type": "https://example.com/api/problems/invalid-request",
  "title": "Invalid sort direction specified",
  "detail": "The sort direction specified must be one of [ ASC, DESC ]"
}
```

This approach to error handling allows you to be as granular or as generic as you like with regards to how errors are handled. The shipped error handler takes an all-or-nothing approach, handling both PHP errors and exceptions/throwables, but treating them all the same. By sprinkling more specific error handlers into your routed middleware pipelines, you can have more control over how your application behaves, based on the context in which it executes.

> While this article demonstrates an approach to building error middleware for reporting in Problem Details format, you will likely want to check out the official offering via the zendframework/zend-problem-details package. We detail that in the chapter REST Representations for Expressive.

Footnotes

1. https://docs.zendframework.com/zend-expressive/features/error-handling/ ↩

2. https://tools.ietf.org/html/rfc7807 ↩

Using Configuration-Driven Routes in Expressive

by Matthew Weier O'Phinney

Expressive 1 used configuration-driven pipelines and routing; Expressive 2 switches to use programmatic pipelines and routes instead. The programmatic approach was chosen as many developers have indicated they find it easier to understand and easier to read, and ensures they do not have any configuration conflicts.

However, there are times you may *want* to use configuration. For example, when you are writing re-usable modules, it's often easier to provide *configuration* for routed middleware, than to expect users to cut-and-paste examples, or use features such as delegator factories[1].

Fortunately, starting in Expressive 2, we offer a couple different mechanisms to support configuration-driven pipelines and routing.

Configuration Only

By default in Expressive 2, and if you run the `expressive-pipeline-from-config` tool to migrate from v1 to v2, we enable a specific flag to force usage of programmatic pipelines:

```php
// Within config/autoload/zend-expressive.global.php in v2,
// and config/autoload/programmatic-pipeline.global.php for v1 projects that
// migrate using the tooling:
return [
    'zend-expressive' => [
        'programmatic_pipeline' => true,
    ],
];
```

By removing this setting, or toggling it to `false`, you can go back to the original Expressive 1 behavior whereby the pipeline and routing are completely generated by configuration. You can read the documentation on the ApplicationFactory[2] for details on how to configure the pipeline and routes in this situation.

> ### Beware!
>
> If you *also* have programmatic declarations in your `config/pipeline.php` and/or `config/routes.php` files, and these are still included from your `public/index.php`, you may run into conflicts when you disable programmatic pipelines! Comment out the `require` lines in your `public/index.php` after toggling the configuration value to be safe!

The key advantage to using configuration is that you can *override* configuration by providing `config/autoload/*.local.php` files; this gives the ability to substitute different middleware when desired. That said, if you use arrays of middleware to create custom pipelines, configuration overriding may not work as expected.

Selective Configuration

There are a few drawbacks to going configuration-only:

- Most pipelines will be static.
- Configuration is more verbose than programmatic declarations.

Fortunately, starting with Expressive 2, you can combine the two approaches, due to the addition of two methods to `Zend\Expressive\Application`:

```php
public function injectPipelineFromConfig(array $config = null) : void;
public function injectRoutesFromConfig(array $config = null) : void;
```

(In each case, if passed no values, they will use the `config` service composed in the container the `Application` instance uses.)

In the case of `injectPipelineFromConfig()`, the method pulls the `middleware_pipeline` value from the passed configuration; `injectRoutesFromConfig()` pulls from the `routes` value.

Where would you use this?

One place to use it is when *modules* provide routing in their `ConfigProvider`. For instance, let's say I have a `BooksApi\ConfigProvider` class that returns a `routes` key with the default set of routes I feel should be defined:

```php
<?php
// in src/BooksApi/ConfigProvider.php:

namespace BooksApi;

class ConfigProvider
```

```
{
    public function __invoke() : array
    {
        return [
            'dependencies' => $this->getDependencies(),
            'routes'       => $this->getRoutes(),
        ];
    }

    public function getDependencies() : array
    {
        // ...
    }

    public function getRoutes() : array
    {
        return [
            [
                'name'            => 'books'
                'path'            => '/api/books',
                'middleware'      => Action\ListBooks::class,
                'allowed_methods' => ['GET'],
            ],
            [
                'path'            => '/api/books',
                'middleware'      => Action\CreateBook::class,
                'allowed_methods' => ['POST'],
            ],
            [
                'name'            => 'book'
                'path'            => '/api/books/{id:\d+}',
                'middleware'      => Action\DisplayBook::class,
                'allowed_methods' => ['GET'],
            ],
            [
                'path'            => '/api/books/{id:\d+}',
                'middleware'      => Action\UpdateBook::class,
                'allowed_methods' => ['PATCH'],
            ],
            [
                'path'            => '/api/books/{id:\d+}',
                'middleware'      => Action\DeleteBook::class,
                'allowed_methods' => ['DELETE'],
            ],
        ];
    }
}
```

If I, as an application developer, feel those defaults do not conflict with my application, I could do the following within my `config/routes.php` file:

```php
<?php
// config/routes.php:

$app->get('/', App\HomePageAction::class, 'home');

$app->injectRoutesFromConfig((new BooksApi\ConfigProvider())());

// ...
```

By invoking the `BooksApi\ConfigProvider`, I can be assured I'm only injecting those routes defined by that given module, and not *all* routes defined *anywhere* in my configuration. I've also saved myself a fair bit of copy-pasta!

> ## Caution: pipelines
>
> We **do not** recommend mixing programmatic and configuration-driven **pipelines**, due to issues of ordering.
>
> When you create a programmatic pipeline, the pipeline is created in exactly the order in which you declare it:
>
> ```php
> $app->pipe(OriginalMessages::class);
> $app->pipe(XClacksOverhead::class);
> $app->pipe(ErrorHandler::class);
> $app->pipe(ServerUrlMiddleware::class);
> $app->pipeRoutingMiddleware();
> $app->pipe(ImplicitHeadMiddleware::class);
> $app->pipe(ImplicitOptionsMiddleware::class);
> $app->pipe(UrlHelperMiddleware::class);
> $app->pipeDispatchMiddleware();
> $app->pipe(NotFoundHandler::class);
> ```
>
> In other words, when you look at the pipeline, you know immediately what the outermost middleware is, and the path to the innermost middleware.
>
> Configuration-driven middleware allows you to specify *priority* values to specify the order in which middleware is piped; higher values are piped earlies, lowest (including negative!) values are piped last.
>
> What happens when you mix the systems? It depends on when you inject configuration-driven middleware:

> ```
> // Middleware from configuration applies first:
> $app->injectPipelineFromConfig($pipelineConfig)
> $app->pipe(/* ... */);
>
> // Middleware from configuration applies last:
> $app->pipe(/* ... */);
> $app->injectPipelineFromConfig($pipelineConfig)
>
> // Or mix it up?
> $app->pipe(/* ... */);
> $app->injectPipelineFromConfig($pipelineConfig)
> $app->pipe(/* ... */);
> ```
>
> This can lead to some tricky situations. We suggest sticking to one or the other, to ensure you can fully visualize the entire pipeline at once.

Summary

The new `Application::injectRoutesFromConfig()` method offered in Expressive 2 provides you with a useful tool for providing routing within your Expressive modules.

This is not the *only* way to provide routing, however. We detail another approach to autowiring routes in the manual[3] that provides a way to keep the programmatic approach, by decorating instantiation of the `Application` instance.

We hope this opens some creative routing possibilities for Expressive developers, particularly those creating reusable modules!

Footnotes

> 1. https://docs.zendframework.com/zend-expressive/cookbook/autowiring-routes-and-pipelines/#delegator-factories ↩

> 2. https://docs.zendframework.com/zend-expressive/features/container/factories/#applicationfactory ↩

> 3. https://docs.zendframework.com/zend-expressive/cookbook/autowiring-routes-and-pipelines/ ↩

Handling OPTIONS and HEAD Requests with Expressive

by Matthew Weier O'Phinney

In v1 releases of Expressive, if you did not define routes that included the `OPTIONS` or `HEAD` HTTP request methods, routing would result in `404 Not Found` statuses, even if a specified route matched the given URI. RFC 7231[1], however, states that both of these request methods SHOULD work for a given resource URI, so long as it exists on the server. This left users in a bit of a bind: if they wanted to comply with the specification (which is often necessary to work correctly with HTTP client software), they would need to either:

- inject additional routes for handling these methods, or
- overload existing middleware to also accept these methods.

In the case of a `HEAD` request, the specification indicates that the resulting response should be identical to that of a `GET` request to the same URI, only with no body content. This would mean having the same response headers.

In the case of an `OPTIONS` request, typically you would respond with a `200 OK` response status, and at least an `Allow` header indicating what HTTP request methods the resource allows.

Sounds like these could be automated, doesn't it?

In Expressive 2, we did!

Handling HEAD requests

If you are using the v2 release of the Expressive skeleton, or have used the `expressive-pipeline-from-config` tool to migrate your application to v2, then you already have support for implicitly adding `HEAD` support to your routes. If not, please go read the documentation[2].

As noted in the documentation, the support is provided by `Zend\Expressive\Middleware\ImplicitHeadMiddleware`, and it operates:

- If the request method is `HEAD`, AND
- the request composes a `RouteResult` attribute, AND
- the route result composes a `Route` instance, AND
- the route returns `true` for the `implicitHead()` method, THEN
- the middleware will return a response.

When the matched route supports the `GET` method, it will dispatch it, and then inject the returned response with an empty body before returning it; this preserves the original response headers, allowing it to operate per RFC 7231 as described above. If `GET` is *not* supported, it simply returns an empty response.

What if you want to customize what happens when `HEAD` is called for a given route?

That's easy: register custom middleware! As a simple, inline example:

```
// In config/routes.php:

use Interop\Http\ServerMiddleware\MiddlewareInterface;
use Interop\Http\ServerMiddleware\DelegateInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\EmptyResponse;

$app->route(
    '/foo',
    new class implements MiddlewareInterface
    {
        public function process(ServerRequestInterface $request, DelegateInterface $de
legate)
        {
            // Return a custom, empty response
            $response = new EmptyResponse(200, [
                'X-Foo' => 'Bar',
            ]);
        }
    },
    ['HEAD']
);
```

Handling OPTIONS requests

Like `HEAD` requests above, if you're using Expressive 2, the middleware for implicitly handling `OPTIONS` requests is already enabled; if not, please go read the documentation[3].

`OPTIONS` requests are handled by `Zend\Expressive\Middleware\ImplicitOptionsMiddleware`, which:

- If the request method is `OPTIONS`, AND
- the request composes a `RouteResult` attribute, AND
- the route result composes a `Route` instance, AND
- the route returns true for the `implicitOptions()` method, THEN
- the middleware will return a response with an `Allow` header indicating methods the route allows.

The Expressive contributors worked to ensure this is consistent across supported router implementations; be aware, however, that if you are using a custom router, it's possible that this may result in `Allow` headers that only contain a subset of all allowed HTTP methods.

What happens if you want to provide a custom `OPTIONS` response? For example, a number of prominent API developers suggest having `OPTIONS` payloads with usage instructions, such as this:

```
HTTP/1.1 200 OK
Allow: GET, POST
Content-Type: application/json

{
    "GET": {
        "query": {
            "page": "int; page of results to return",
            "per_page": "int; number of results to return per page"
        },
        "response": {
            "total": "Total number of items",
            "count": "Total number of items returned on this page",
            "_links": {
                "self": "URI to collection",
                "first": "URI to first page of results",
                "prev": "URI to previous page of results",
                "next": "URI to next page of results",
                "last": "URI to last page of results",
                "search": "URI template for searching"
            },
            "_embedded": {
                "books": [
                    "See ... for details"
                ]
            }
        }
    },
    "POST": {
        "data": {
            "title": "string; title of book",
            "author": "string; author of book",
            "info": "string; book description and notes"
        },
        "response": {
            "_links": {
                "self": "URI to book"
            },
            "id": "string; generated UUID for book",
            "title": "string; title of book",
            "author": "string; author of book",
            "info": "string; book description and notes"
        }
    }
}
```

The answer is the same as with `HEAD` requests: register a custom route!

```
<?php
// In config/routes.php:
```

```
use Interop\Http\ServerMiddleware\MiddlewareInterface;
use Interop\Http\ServerMiddleware\DelegateInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\JsonResponse;

$app->route(
    '/books',
    new class implements MiddlewareInterface
    {
        public function process(ServerRequestInterface $request, DelegateInterface $de
legate)
        {
            // Return a custom response
            $response = new JsonResponse([
                'GET' => [
                    'query' => [
                        'page' => 'int; page of results to return',
                        'per_page' => 'int; number of results to return per page',
                    ],
                    'response' => [
                        'total' => 'Total number of items',
                        'count' => 'Total number of items returned on this page',
                        '_links' => [
                            'self' => 'URI to collection',
                            'first' => 'URI to first page of results',
                            'prev' => 'URI to previous page of results',
                            'next' => 'URI to next page of results',
                            'last' => 'URI to last page of results',
                            'search' => 'URI template for searching',
                        ],
                        '_embedded' => [
                            'books' => [
                                'See ... for details',
                            ],
                        ],
                    ],
                ],
                'POST' => [
                    'data' => [
                        'title' => 'string; title of book',
                        'author' => 'string; author of book',
                        'info' => 'string; book description and notes',
                    ],
                    'response' => [
                        '_links' => [
                            'self' => 'URI to book',
                        ],
                        'id' => 'string; generated UUID for book',
                        'title' => 'string; title of book',
                        'author' => 'string; author of book',
                        'info' => 'string; book description and notes',
                    ],
                ],
            ], 200, ['Allow' => 'GET,POST']);
        }
    },
    ['OPTIONS']
);
```

Final word

Obviously, you may not want to use inline classes as described above, but hopefully with the above examples, you can begin to see the possibilities for handling `HEAD` and `OPTIONS` requests in Expressive. The simplest option, which will likely suffice for the majority of use cases, is now built-in to the skeleton, and added by default when using the migration tools. For those other cases where you need further customization, Expressive's routing capabilities give you the flexibility and power to accomplish whatever you might need.

For more information on the built-in capabilities, visit the documentation[4].

Footnotes

> [1]. https://tools.ietf.org/html/rfc7231 ↩

> [2]. https://docs.zendframework.com/zend-expressive/features/middleware/implicit-methods-middleware/#implicitheadmiddleware ↩

> [3]. https://docs.zendframework.com/zend-expressive/features/middleware/implicit-methods-middleware/#implicitoptionsmiddleware ↩

> [4]. https://docs.zendframework.com/zend-expressive/features/middleware/implicit-methods-middleware/ ↩

Caching middleware with Expressive

by Enrico Zimuel

Performance is one of the key feature for web application. Using a middleware architecture makes it very simple to implement a caching system in PHP.

The general idea is to store the response output of a URL in a file (or in memory, using memcached[1]) and use it for subsequent requests. In this way we can bypass the execution of middleware nested further in the pipeline starting from the second request.

Of course, this technique can only be applied for static content that does not change between requests.

Implement a caching middleware

Imagine we want to create a simple cache system with Expressive. We can use an implementation like the following:

```
namespace App\Action;

use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface as ServerMiddlewareInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\HtmlResponse;

class CacheMiddleware implements ServerMiddlewareInterface
{
    private $config;

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

    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        $url  = str_replace('/', '_', $request->getUri()->getPath());
        $file = $this->config['path'] . $url . '.html';
        if ($this->config['enabled'] && file_exists($file) &&
            (time() - filemtime($file)) < $this->config['lifetime']) {
            return new HtmlResponse(file_get_contents($file));
        }

        $response = $delegate->process($request);
        if ($response instanceof HtmlResponse && $this->config['enabled']) {
            file_put_contents($file, $response->getBody());
        }
        return $response;
    }
}
```

The idea of this middleware is quite simple. If the caching system is enabled and if the requested URL matches an existing cache file, we return the cache content as an HtmlResponse[2], ending the execution flow.

If the requested URL path does not exist in cache, we delegate to the next middleware in our queue, and, if caching is enabled, cache the response before returning it.

Configuring the cache system

To manage the cache, we used a configuration key `cache` to specify the `path` of the cache files, the `lifetime` in seconds and the `enabled` value to turn the caching system on and off.

Since we use a file to store the cache content, we can use the file modification time to manage the lifetime of the cache via the PHP function `filemtime()` [3].

> Note: if you want to use memcached instead of the filesystem, you need to replace the `file_get_contents()` and `file_put_contents()` functions with `Memcached::get()` and `Memcached::set()`. Moreover, you do not need to check for lifetime because when you will instead set an expiration time when pushing content to memcached.

In order to pass the `$config` dependency, we will create a factory class:

```php
namespace App\Action;

use Interop\Container\ContainerInterface;
use Exception;

class CacheFactory
{
    public function __invoke(ContainerInterface $container)
    {
        $config = $container->get('config');
        $config = $config['cache'] ?? [];

        if (! array_key_exists('enabled', $config)) {
            $config['enabled'] = false;
        }

        if ($config['enabled']) {
            if (! isset($config['path'])) {
                throw new Exception('The cache path is not configured');
            }
            if (! isset($config['lifetime'])) {
                throw new Exception('The cache lifetime is not configured');
            }
        }

        return new CacheMiddleware($config);
    }
}
```

We can store this configuration in a plain PHP file in the `config/autoload/` directory; for example, we could put it in `config/autoload/cache.local.php` with the following contents:

```php
return [
    'cache' => [
      'enabled'  => true,
      'path'     => 'data/cache/',
      'lifetime' => 3600 // in seconds
    ]
];
```

In the above, we specify `data/cache/` as the cache directory; since Expressive sets the working directory to the application root, this will, resolve to that location.

The content of this folder should be omitted from your version control; as an example, with Git, you can place a `.gitignore` file inside the `cache` folder with the following content:

```
*
!.gitignore
```

Next, in order to activate the caching system, we need to add the `CacheMiddleware` class as service. We do that via a global configuration file such as `/config/autoload/cache.global.php` with the following content:

```
return [
    'dependencies' => [
        'factories' => [
            App\Action\CacheMiddleware::class => App\Action\CacheFactory::class
        ]
    ]
];
```

Enabling caching for specific routes

We mentioned earlier that this caching mechanism only works for static content. That means we need a way to enable the cache only for specific routes. We do this by specifying an array of middleware for those routes, and adding the `CacheMiddleware` class as first middleware to be executed in those routes.

For instance, imagine we have an `/about` route that displays an "About" page for your website. We can add the `CacheMiddleware` as follows:

```
use App\Action;

$app->get('/about', [
    Action\CacheMiddleware::class,
    Action\AboutAction::class
], 'about');
```

Because the `CacheMiddleware` appears first in the array, it will execute first, delivering the cached contents after the first request to the route. (The `$app` object is an instance of `Zend\Expressive\Application`.)

Conclusion

In this article, we demonstrated building a lightweigth caching system for a PHP middleware application. A middleware architecture facilitates the design of a cache layer because it allows composing middleware to create an HTTP request workflow.

Footnotes

> 1. https://memcached.org ↩

> 2. https://docs.zendframework.com/zend-diactoros/custom-responses/#html-responses ↩

> 3. http://php.net/filemtime ↩

Middleware authentication

by Enrico Zimuel

Many web applications require restricting specific areas to *authenticated* users, and may further restrict specific actions to *authorized* user roles. Implementing authentication and authorization in a PHP application is often non-trivial as doing so requires altering the application workflow. For instance, if you have an MVC design, you may need to change the dispatch logic to add an authentication layer as an initial event in the execution flow, and perhaps apply restrictions within your controllers.

Using a middleware approach is simpler and more natural, as middleware easily accommodates workflow changes. In this article, we will demonstrate how to provide authentication in a PSR-7 middleware application using Expressive and zend-authentication[1]. We will build a simple authentication system using a login page with username and password credentials.

> We detail authorization in the next article.

Getting started

This article assumes you have already created an Expressive application. For the purposes of our application, we'll create a new module, `Auth`, in which we'll put our classes, middleware, and general configuration.

First, if you have not already, install the tooling support:

```
$ composer require --dev zendframework/zend-expressive-tooling
```

Next, we'll create the `Auth` module:

```
$ ./vendor/bin/expressive module:create Auth
```

With that out of the way, we can get started.

Authentication

The zend-authentication component offers an adapter-based authentication solution, with both a number of concrete adapters as well as mechanisms for creating and consuming custom adapters.

The component exposes `Zend\Authentication\Adapter\AdapterInterface` , which defines a single `authenticate()` method:

```
namespace Zend\Authentication\Adapter;

interface AdapterInterface
{
    /**
     * Performs an authentication attempt
     *
     * @return \Zend\Authentication\Result
     * @throws Exception\ExceptionInterface if authentication cannot be performed
     */
    public function authenticate();
}
```

Adapters implementing the `authenticate()` method perform the logic necessary to authenticate a request, and return the results via a `Zend\Authentication\Result` object. This `Result` object contains the authentication result code and, in the case of success, the user's identity. The authentication result codes are defined using the following constants:

```
namespace Zend\Authentication;

class Result
{
    const SUCCESS = 1;
    const FAILURE = 0;
    const FAILURE_IDENTITY_NOT_FOUND = -1;
    const FAILURE_IDENTITY_AMBIGUOUS = -2;
    const FAILURE_CREDENTIAL_INVALID = -3;
    const FAILURE_UNCATEGORIZED = -4;
}
```

If we want to implement a login page with `username` and `password` authentication, we can create a custom adapter such as the following:

```php
// In src/Auth/src/MyAuthAdapter.php:

namespace Auth;

use Zend\Authentication\Adapter\AdapterInterface;
use Zend\Authentication\Result;

class MyAuthAdapter implements AdapterInterface
{
    private $password;
    private $username;

    public function __construct(/* any dependencies */)
    {
        // Likely assign dependencies to properties
    }

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

    public function setUsername(string $username) : void
    {
        $this->username = $username;
    }

    /**
     * Performs an authentication attempt
     *
     * @return Result
     */
    public function authenticate()
    {
        // Retrieve the user's information (e.g. from a database)
        // and store the result in $row (e.g. associative array).
        // If you do something like this, always store the passwords using the
        // PHP password_hash() function!

        if (password_verify($this->password, $row['password'])) {
            return new Result(Result::SUCCESS, $row);
        }

        return new Result(Result::FAILURE_CREDENTIAL_INVALID, $this->username);
    }
}
```

We will want a factory for this service as well, so that we can seed the username and password to it later:

```
// In src/Auth/src/MyAuthAdapterFactory.php:

namespace Auth;

use Interop\Container\ContainerInterface;
use Zend\Authentication\AuthenticationService;

class MyAuthAdapterFactory
{
    public function __invoke(ContainerInterface $container)
    {
        // Retrieve any dependencies from the container when creating the instance
        return new MyAuthAdapter(/* any dependencies */);
    }
}
```

This factory class creates and returns an instance of `MyAuthAdapter`. We may need to pass some dependencies to its constructor, such as a database connection; these would be fetched from the container.

Authentication Service

We can now create a `Zend\Authentication\AuthenticationService` that composes our adapter, and then consume the `AuthenticationService` in middleware to check for a valid user. Let's now create a factory for the `AuthenticationService`:

```
// in src/Auth/src/AuthenticationServiceFactory.php:

namespace Auth;

use Interop\Container\ContainerInterface;
use Zend\Authentication\AuthenticationService;

class AuthenticationServiceFactory
{
    public function __invoke(ContainerInterface $container)
    {
        return new AuthenticationService(
            null,
            $container->get(MyAuthAdapter::class)
        );
    }
}
```

This factory class retrieves an instance of the `MyAuthAdapter` service and use it to return an `AuthenticationService` instance. The `AuthenticationService` class accepts two parameters:

- A storage service instance, for persisting the user identity. If none is provided, the built-in PHP session mechanisms will be used.
- The actual adapter to use for authentication.

Now that we have created both the custom adapter, as well as factories for the adapter and the `AuthenticationService`, we need to configure our application dependencies to use them:

```php
// In src/Auth/src/ConfigProvider.php:

// Add the following import statement at the top of the classfile:
use Zend\Authentication\AuthenticationService;

// And update the following method:
public function getDependencies()
{
    return [
        'factories' => [
            AuthenticationService::class => AuthenticationServiceFactory::class,
            MyAuthAdapter::class => MyAuthAdapterFactory::class,
        ],
    ];
}
```

Authenticate using a login page

With an authentication mechanism in place, we now need to create middleware to render the login form. This middleware will do the following:

- for `GET` requests, it will render the login form.
- for `POST` requests, it will check for credentials and then attempt to validate them.
    - for valid authentication requests, we will redirect to a welcome page
    - for invalid requests, we will provide an error message and redisplay the form.

Let's create the middleware now:

```php
// In src/Auth/src/Action/LoginAction.php:

namespace Auth\Action;

use Auth\MyAuthAdapter;
use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface as ServerMiddlewareInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Authentication\AuthenticationService;
use Zend\Diactoros\Response\HtmlResponse;
use Zend\Diactoros\Response\RedirectResponse;
use Zend\Expressive\Template\TemplateRendererInterface;
```

```
class LoginAction implements ServerMiddlewareInterface
{
    private $auth;
    private $authAdapter;
    private $template;

    public function __construct(
        TemplateRendererInterface $template,
        AuthenticationService $auth,
        MyAuthAdapter $authAdapter
    ) {
        $this->template    = $template;
        $this->auth        = $auth;
        $this->authAdapter = $authAdapter;
    }

    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        if ($request->getMethod() === 'POST') {
            return $this->authenticate($request);
        }

        return new HtmlResponse($this->template->render('auth::login'));
    }

    public function authenticate(ServerRequestInterface $request)
    {
        $params = $request->getParsedBody();

        if (empty($params['username'])) {
            return new HtmlResponse($this->template->render('auth::login', [
                'error' => 'The username cannot be empty',
            ]));
        }

        if (empty($params['password'])) {
            return new HtmlResponse($this->template->render('auth::login', [
                'username' => $params['username'],
                'error'    => 'The password cannot be empty',
            ]));
        }

        $this->authAdapter->setUsername($params['username']);
        $this->authAdapter->setPassword($params['password']);

        $result = $this->auth->authenticate();
        if (!$result->isValid()) {
            return new HtmlResponse($this->template->render('auth::login', [
                'username' => $params['username'],
                'error'    => 'The credentials provided are not valid',
            ]));
        }

        return new RedirectResponse('/admin');
    }
}
```

This middleware manages two actions: rendering the login form, and authenticating the user's credentials when submitted via a `POST` request.

> You will also need to ensure that you have:
>
> - Created a `login` template.
> - Added configuration to map the `auth` template namespace to one or more filesystem paths.
>
> We leave those tasks as an exercise to the reader.

We now need to create a factory to provide the dependencies for this middleware:

```
// In src/Auth/src/Action/LoginActionFactory.php:

namespace Auth\Action;

use Auth\MyAuthAdapter;
use Interop\Container\ContainerInterface;
use Zend\Authentication\AuthenticationService;
use Zend\Expressive\Template\TemplateRendererInterface;

class LoginActionFactory
{
    public function __invoke(ContainerInterface $container)
    {
        return new LoginAction(
            $container->get(TemplateRendererInterface::class),
            $container->get(AuthenticationService::class),
            $container->get(MyAuthAdapter::class)
        );
    }
}
```

Map the middleware to this factory in your dependencies configuration witin the `ConfigProvider`:

```
// In src/Auth/src/ConfigProvider.php,

// Update the following method to read as follows:
public function getDependencies()
{
    return [
        'factories' => [
            Action\LoginAction::class => Action\LoginActionFactory::class,
            AuthenticationService::class => AuthenticationServiceFactory::class,
            MyAuthAdapter::class => MyAuthAdapterFactory::class,
        ],
    ];
}
```

> ### Use zend-servicemanager's ReflectionBasedAbstractFactory
>
> If you are using zend-servicemanager in your application, you could skip the step of creating the factory, and instead map the middleware to `Zend\ServiceManager\AbstractFactory\ReflectionBasedAbstractFactory`.

Finally, we can create appropriate routes. We'll map `/login` to the `LoginAction` now, and allow it to react to either the `GET` or `POST` methods:

```
// in config/routes.php:
$app->route('/login', Auth\Action\LoginAction::class, ['GET', 'POST'], 'login');
```

Alternately, the above could be written as two separate statements:

```
// in config/routes.php:
$app->get('/login', Auth\Action\LoginAction::class, 'login');
$app->post('/login', Auth\Action\LoginAction::class);
```

Authentication middleware

Now that we have the authentication service and its adapter and the login middleware in place, we can create middleware that checks for authenticated users, having it redirect to the `/login` page if the user is not authenticated.

```php
// In src/Auth/src/Action/AuthAction.php:

namespace Auth\Action;

use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface as ServerMiddlewareInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Authentication\AuthenticationService;
use Zend\Diactoros\Response\RedirectResponse;

class AuthAction implements ServerMiddlewareInterface
{
    private $auth;

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

    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        if (! $this->auth->hasIdentity()) {
            return new RedirectResponse('/login');
        }

        $identity = $this->auth->getIdentity();
        return $delegate->process($request->withAttribute(self::class, $identity));
    }
}
```

This middleware checks for a valid identity using the `hasIdentity()` method of `AuthenticationService`. If no identity is present, we redirect the `redirect` configuration value.

If the user is authenticated, we continue the execution of the next middleware, storing the identity in a request attribute. This facilitates consumption of the identity information in subsequent middleware layers. For instance, imagine you need to retrieve the user's information:

```
namespace App\Action;

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

class FooAction
{
    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        $user = $request->getAttribute(AuthAction::class);
        // $user will contains the user's identity
    }
}
```

The `AuthAction` middleware needs some dependencies, so we will need to create and register a factory for it as well.

First, the factory:

```
// In src/Auth/src/Action/AuthActionFactory.php:

namespace Auth\Action;

use Interop\Container\ContainerInterface;
use Zend\Authentication\AuthenticationService;
use Exception;

class AuthActionFactory
{
    public function __invoke(ContainerInterface $container)
    {
        return new AuthAction($container->get(AuthenticationService::class));
    }
}
```

And then mapping it:

```
// In src/Auth/src/ConfigProvider.php:


// Update the following method to read as follows:
public function getDependencies()
{
    return [
        'factories' => [
            Action\AuthAction::class => Action\AuthActionFactory::class,
            Action\LoginAction::class => Action\LoginActionFactory::class,
            AuthenticationService::class => AuthenticationServiceFactory::class,
            MyAuthAdapter::class => MyAuthAdapterFactory::class,
        ],
    ];
}
```

> Like the `LoginActionFactory` above, you could skip the factory creation and instead use the `ReflectionBasedAbstractFactory` if using zend-servicemanager.

Require authentication for specific routes

Now that we built the authentication middleware, we can use it to protect specific routes that require authentication. For instance, for each route that needs authentication, we can modify the routing to create a pipeline that incorporates our `AuthAction` middleware early:

```
$app->get('/admin', [
    Auth\Action\AuthAction::class,
    App\Action\DashBoardAction::class
], 'admin');

$app->get('/admin/config', [
    Auth\Action\AuthAction::class,
    App\Action\ConfigAction::class
], 'admin.config');
```

The order of execution for the middleware is the order of the array elements. Since the `AuthAction` middleware is provided as the first element, if a user is not authenticated when requesting either the admin dashboard or config page, they will be immediately redirected to the login page instead.

Conclusion

There are many ways to accommodate authentication within middleware applications; this is just one. Our goal was to demonstrate the ease with which you may compose authentication into existing workflows by creating middleware that intercepts the request early within a pipeline.

You could certainly make a number of improvements to the workflow:

- The path to the login page could be configurable.
- You could capture the original request path in order to allow redirecting to it following successful login.
- You could introduce rate limiting of login requests.

These are each interesting exercises for you to try!

Footnotes

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

Authorize users using Middleware

by Enrico Zimuel

In the previous article, we demonstrated how to authenticate a middleware application in PHP. In this post we will continue the discussion, showing how to manage *authorization*.

We will start from an authenticated user and demonstrate how to allow or disable actions for specific users. We will collect users by groups and we will use a *Role-Based Access Control* (RBAC) system to manage the authorizations.

To implement RBAC, we will consume zendframework/zend-permissions-rbac[1].

> If you are not familiar with RBAC and the usage of zend-permissions-rbac, we cover the topic on our blog[2], as well as in the Zend Framework 3 Cookbook.

Getting started

This article assumes you have already created the `Auth` module, as described in our previous article on authentication. For the purposes of our application, we'll create a new module, `Permission`, in which we'll put our classes, middleware, and general configuration.

First, if you have not already, install the tooling support:

```
$ composer require --dev zendframework/zend-expressive-tooling
```

Next, we'll create the `Permission` module:

```
$ ./vendor/bin/expressive module:create Permission
```

With that out of the way, we can get started.

Authentication

As already mentioned, we will reuse the `Auth` module created in our previous post. We will reuse the `Auth\Action\AuthAction::class` to get the authenticated user's data.

Authorization

In order to manage authorization, we will use a RBAC system using the user's role. A user's *role* is a string that represents the permission level; as an example, the role `administrator` might provide access to all permissions.

In our scenario, we want to allow or disable access of specific routes to a role or set of roles. Each route represents a *permission* in RBAC terminology.

We can use zendframework/zend-permissions-rbac[3] to manage the RBAC system using a PHP configuration file storing the list of roles and permissions. Using zend-permissions-rbac, we can also manage permissions inheritance.

For instance, imagine implementing a blog application; we might define the following roles:

- `administrator`
- `editor`
- `contributor`

A `contributor` can create, edit, and delete only the posts created by theirself. The `editor` can create, edit, and delete all posts and publish posts (that means enabling public view of a post in the web site). The `administrator` can perform all actions, including changing the blog's configuration.

This is a perfect use case for using permission inheritance. In fact, the `administrator` role would inherit the permissions of the `editor`, and the `editor` role inherits the permissions of the `contributor`.

To manage the previous scenario, we can use the following configuration file:

```
// In src/Permission/config/rbac.php:

return [
    'roles' => [
        'administrator' => [],
        'editor'        => ['admin'],
        'contributor'   => ['editor'],
    ],
    'permissions' => [
        'contributor' => [
            'admin.dashboard',
            'admin.posts',
        ],
        'editor' => [
            'admin.publish',
        ],
        'administrator' => [
            'admin.settings',
        ],
    ],
];
```

In this file we have specified three roles, including the inheritance relationship using an array of role names. The parent of `administator` is an empty array, meaning no parents.

The permissions are configured using the `permissions` key. Each role has the list of permissions, specified with an array of route names.

All the roles can access the route `admin.dashboard` and `admin.posts`. The `editor` role can also access `admin.publish`. The `administrator` can access all the roles of `contributor` and `editor`. Moreover, only the `administrator` can access the `admin.settings` route.

> We used the route names as RBAC permissions because in this way we can allow URL and HTTP methods using a single resource name. Moreover, in Expressive we have a `config/routes.php` file containing all the routes and we can easily use it to add authorization, as we did for authentication.

Authorization middleware

Now that we have the RBAC configuration in place, we can create a middleware that performs the user authorization verifications.

We can create an `AuthorizationAction` middleware in our `Permission` module as follows:

```
// in src/Permission/src/Action/AuthorizationAction.php:
```

```
namespace Permission\Action;

use Auth\Action\AuthAction;
use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface as MiddlewareInterface;
use Permission\Entity\Post as PostEntity;
use Permission\Service\PostService;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Diactoros\Response\EmptyResponse;
use Zend\Expressive\Router\RouteResult;
use Zend\Permissions\Rbac\AssertionInterface;
use Zend\Permissions\Rbac\Rbac;
use Zend\Permissions\Rbac\RoleInterface;

class AuthorizationAction implements MiddlewareInterface
{
    private $rbac;
    private $postService;

    public function __construct(Rbac $rbac, PostService $postService)
    {
        $this->rbac        = $rbac;
        $this->postService = $postService;
    }

    public function process(ServerRequestInterface $request, DelegateInterface $delega
te)
    {
        $user = $request->getAttribute(AuthAction::class, false);
        if (false === $user) {
            return new EmptyResponse(401);
        }

        // if a post attribute is present and user is contributor
        $postUrl = $request->getAttribute('post', false);
        if (false !== $postUrl && 'contributor' === $user['role']) {
            $post = $this->postService->getPost($postUrl);
            $assert = new class ($user['username'], $post) implements AssertionInterfa
ce {
                private $post;
                private $username;

                public function __construct(string $username, PostEntity $post)
                {
                    $this->username = $username;
                    $this->post     = $post;
                }

                public function assert(Rbac $rbac)
                {
                    return $this->username === $this->post->getAuthor();
                }
            };
        }

        $route     = $request->getAttribute(RouteResult::class);
        $routeName = $route->getMatchedRoute()->getName();
        if (! $this->rbac->isGranted($user['role'], $routeName, $assert ?? null)) {
            return new EmptyResponse(403);
        }

        return $delegate->process($request);
    }
}
```

If the user is not present, the `AuthAction::class` attribute will be false. In this case we are returning a `401` error, indicating we have an unauthenticated user, and halting execution.

If a user is returned from `AuthAction::class` attribute, this means that we have an authenticated user.

> The authentication is performed by the `Auth\Action\AuthAction` class that stores the `AuthAction::class` attribute in the request. See the previous article for more information.

This middleware performs the authorization check using `isGranted($role, $permission)` where `$role` is the user's role (`$user['role']`) and `$permission` is the route name, retrieved by the `RouteResult::class` attribute. If the role is granted, we continue the execution flow with the delegate middleware. Otherwise, we stop the execution with a `403` error, indicating lack of authorization.

We manage also the case when the user is a `contributor` and there is a `post` attribute in the request (e.g. /admin/posts/{post}). That means someone is performing some action on a specific post. To perform this action, we require that the owner of the post should be the same as the authenticated user.

This will prevent a `contributor` to change the content of a post if he/she is not the author. We managed this special case using a dynamic assertion[4], built using an anonymous class; it checks if the authenticated `username` is the same of the author's post. We used a general `PostEntity` class with a `getAuthor()` function.

In order to retrieve for the route name, we used the `RouteResult::class` attribute provided by Expressive. This attribute facilitates access to the matched route.

The `AuthorizationAction` middleware requires the `Rbac` and the `PostService` dependencies. The first is an instance of `Zend\Permissions\Rbac\Rbac` and the second is a general service to manage blog posts, i.e. a class that performs some lookup to retrieve the post data from a database.

To inject these dependencies, we use an `AuthorizationFactory` like the following:

```php
namespace Permission\Action;

use Interop\Container\ContainerInterface;
use Zend\Permissions\Rbac\Rbac;
use Zend\Permissions\Rbac\Role;
use Permission\Service\PostService;
use Exception;

class AuthorizationFactory
{
    public function __invoke(ContainerInterface $container)
    {
        $config = $container->get('config');
        if (! isset($config['rbac']['roles'])) {
            throw new Exception('Rbac roles are not configured');
        }
        if (!isset($config['rbac']['permissions'])) {
            throw new Exception('Rbac permissions are not configured');
        }

        $rbac = new Rbac();
        $rbac->setCreateMissingRoles(true);

        // roles and parents
        foreach ($config['rbac']['roles'] as $role => $parents) {
            $rbac->addRole($role, $parents);
        }

        // permissions
        foreach ($config['rbac']['permissions'] as $role => $permissions) {
            foreach ($permissions as $perm) {
                $rbac->getRole($role)->addPermission($perm);
            }
        }
        $post = $container->get(PostService::class);

        return new AuthorizationAction($rbac, $post);
    }
}
```

This factory class builds the `Rbac` object using the configuration file stored in `src/Permission/config/rbac.php`. We read all the roles and the permissions following the order in the array. It is important to enable the creation of missing roles in the Rbac object using the function `setCreateMissingRoles(true)`. This is required to be sure to create all the roles even if we add it out of order. For instance, without this setting, the following configuration will throw an exception:

```
return [
    'roles' => [
        'contributor'   => ['editor'],
        'editor'        => ['administrator'],
        'administrator' => [],
    ],
];
```

because the `editor` and the `administrator` roles are specified as parents of other roles before they were created.

Finally, we can configure the `Permission` module adding the following dependencies:

```
// In src/Permission/src/ConfigProvider.php:

// Update the following methods:
public function __invoke()
{
    return [
        'dependencies' => $this->getDependencies(),
        'rbac'         => include __DIR__ . '/../config/rbac.php',
    ];
}

public function getDependencies()
{
    return [
        'factories' => [
            Service\PostService::class => Service\PostFactory::class,
            Action\AuthorizationAction::class => Action\AuthorizationFactory::class,
        ],
    ];
}
```

Configure the route for authorization

To enable authorization on a specific route, we need to add the `Permission\Action\AuthorizationAction` middleware in the route, as follows:

```
$app->get('/admin/dashboard', [
    Auth\Action\AuthAction::class,
    Permission\Action\AuthorizationAction::class,
    Admin\Action\DashboardAction::class
], 'admin.dashboard');
```

This is an example of the `GET /admin/dashboard` route with `admin.dashboard` as the name. We add `AuthAction` and `AuthorizationAction` before execution of the `DashboardAction`. The order of the middleware array is important; authentication must happen first, and authorization must happen before executing the dashboard middleware.

Add the `AuthorizationAction` middleware to all routes requiring authorization.

Conclusion

This article, together with the previous, demonstrates how to accomodate authentication and authorization within middleware in PHP.

We demonstated how to create two separate Expressive modules, `Auth` and `Permission`, to provide authentication and authorization using zend-authentication and zend-permissions-rbac.

We also showed the usage of a dynamic assertion for specific permissions based on the role and username of an authenticated user.

The blog use case proposed in this article is quite simple, but the architecture used can be applied also in complex scenarios, to manage permissions based on different requirements.

Footnotes

1. https://docs.zendframework.com/zend-permissions-rbac ↩
2. https://framework.zend.com/blog/2017-04-27-zend-permissions-rbac.html ↩
3. https://docs.zendframework.com/zend-permissions-rbac ↩
4. https://docs.zendframework.com/zend-permissions-rbac/intro/#dynamic-assertions ↩

REST Representations for Expressive

by Matthew Weier O'Phinney

At the time of writing (September 2017), we have published two experimental components for providing REST response representations for middleware applications:

- zend-problem-details: https://github.com/zendframework/zend-problem-details
- zend-expressive-hal: https://github.com/zendframework/zend-expressive-hal

These components provide *response representations* for APIs built with PSR-7[1] middleware. Specifically, they provide:

- Problem Details for HTTP APIs (RFC 7807)[2]
- Hypertext Application Language (HAL)[3]

These two formats provide both JSON and XML representation options (the latter through a secondary proposal[4]).

What's in a representation?

So you're developing an API!

What can clients expect when they make a request to your API? Will they get a wall of text? or some sort of serialization? If it's a serialized format, which ones do you support? And how is the data structured?

The typical answer will be, "we'll provide JSON responses." That answers the serialization aspect, but not the *data structure*; for that, you might develop and publish a schema for your end users, so they know how to parse the response.

But you may still have unanswered questions:

- How does the consumer know what actions can next be taken, or what resources might be related to the one requested?
- If the resource contains other entities, how can they identify which ones they can request separately, versus those that are just part of the data structure?

These and all of the previous are questions that a representation format answers. A well-considered representation format will:

- Provide *links* to the actions that may be performed next, as well as to related resources.
- Indicate which data elements represent other requestable resources.

- Be extensible, to allow representing arbitrary data.

I tend to think of representations as falling into two buckets:

- Representations of errors.
- Representations of application resources.

Errors need separate representation, as they are not *requestable* on their own; they are returned when something goes wrong, and need to provide enough detail that the consumer can determine what they need to change in order to perform a new request.

The Problem Details specification provides exactly this. As an example:

```
{
    "type": "https://example.com/problems/rate-limit-exceeded",
    "title": "You have exceeded your API rate limit.",
    "detail": "You have hit your rate limit of 5000 requests per hour.",
    "requests_this_hour": 5025,
    "rate_limit": 5000,
    "rate_limit_reset": "2017-05-03T14:39-0500"
}
```

We chose Problem Details to standardize on when starting the Apigility project as it has very few requirements, but can model any error easily. The ability to link to documentation detailing general error types provides the ability to communicate with your consumers about known errors and how to correct them.

Application resources generally should have their own schema, but having a predictable structure for providing *relational links* (answering the "what can I do next" question) and embedding related resources can help those making clients or those consuming your API to automate many of their processes. Instead of having a list of URLs they can access, they can hit one resource, and start following the composed links; when they present data, they can also present controls for the embedded resources, making it simpler to make requests to these other items.

HAL provides these details in a simple way: relational links are objects under the `_links` element, and embedded resources are under the `_embedded` element; all other data is represented as normal key/value pairs, allowing for arbitrary nesting of structures. An example payload might look like the following:

```
{
    "_links": {
        "self": { "href": "/api/books?page=7" },
        "first": { "href": "/api/books?page=1" },
        "prev": { "href": "/api/books?page=6" },
        "next": { "href": "/api/books?page=8" },
        "last": { "href": "/api/books?page=17" }
        "search": {
            "href": "/api/books?query={searchTerms}",
            "templated": true
        }
    },
    "_embedded": {
        "book": [
            {
                "_links": {
                    "self": { "href": "/api/books/1234" }
                }
                "id": 1234,
                "title": "Hitchhiker's Guide to the Galaxy",
                "author": "Adams, Douglas"
            },
            {
                "_links": {
                    "self": { "href": "/api/books/6789" }
                }
                "id": 6789,
                "title": "Ancillary Justice",
                "author": "Leckie, Ann"
            }
        ]
    },
    "_page": 7,
    "_per_page": 2,
    "_total": 33
}
```

The above provides controls to allow a consumer to navigate through a result set, as well as to perform another search against the API. It provides data about the result set, and also embeds a number of resources, with links so that the consumer can make requests against those individually. Having links present in the payloads means that if the URI scheme changes later, a well-written client will be unaffected, *as it will follow the links delivered in response payloads* instead of hard-coding them. This allows our API to evolve, without affecting the robustness of clients.

A number of other representation formats have become popular over the years, including:

- JSON API[5]
- Collection+JSON[6]

[7]

- Siren[7]

Each are powerful and flexible in their own right. We standardized on HAL for Apigility originally as it was one of the first published specifications; we've continued with it as it is a format that's both easy to generate as well as parse, and extensible enough to answer the needs of most API representations.

zend-problem-details

The package zendframework/zend-problem-details[8] provides a Problem Details implementation for PHP, and specifically for generating PSR-7 responses. It provides a multi-faceted approach to providing error details to your users.

First, you can compose the `ProblemDetailsResponseFactory` into your middleware, and use it to generate and return your error responses:

```
return $this->problemDetails->createResponse(
    $request,                                               // PSR-7 request
    422,                                                    // HTTP status
    'Invalid data detected in book submission',             // Detail
    'Invalid book',                                         // Problem title
    'https://example.com/api/doc/errors/invalid-book',      // Problem type (URL to detail
s)
    ['messages' => $validator->getMessages()]               // Additional data
);
```

> The request instance is passed to the factory to allow it to perform *content negotiation*; zend-problem-details uses the `Accept` header to determine whether to serve a JSON or an XML representation, defaulting to XML if it is unable to match to either format.

The above will generate a response like the following:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/problem+json

{
  "status": 422,
  "title": "Invalid Book",
  "type": "https://example.com/api/doc/errors/invalid-book",
  "detail": "Invalid data detected in book submission",
  "messages": [
    "Missing title",
    "Missing author"
  ]
}
```

> `ProblemDetailsFactory` is agnostic of PSR-7 implementation, and allows you to inject a response prototype and stream factory during instantiation. By default, it uses zend-diactoros for these artifacts if none are provided.

Second, you can create a response from a caught exception or throwable:

```
return $this->problemDetails->createResponseFromThrowable(
    $request,
    $throwable
);
```

Currently, the factory uses the exception message for the detail, and 4XX and 5XX exception codes for the status (defaulting to 500 for any other value).

> At the time of publication, we are currently evaluating a proposal that would have caught exceptions generate a canned Problem Details response with a status of 500, so the above behavior may change in the future. If you want to guarantee the code and message are used, you can create custom exceptions, as outlined below.

Third, extending on the ability to create details from throwables, we provide a custom exception interface, `ProblemDetailsException`. This interface defines methods for pulling additional information to provide to a Problem Details response:

```
namespace Zend\ProblemDetails\Exception;

interface ProblemDetailsException
{
    public function getStatus() : int;
    public function getType() : string;
    public function getTitle() : string;
    public function getDetail() : string;
    public function getAdditionalData() : array;
}
```

If you throw an exception that implements this interface, the `createResponseFromThrowable()` method shown above will pull data from these methods in order to create the response. This allows you to define domain-specific exceptions that can provide additional details when used in an API context.

Finally, we also provide optional middleware, `ProblemDetailsMiddleware`, that does the following:

- Registers an error handler that casts PHP errors in the current `error_reporting` bitmask to `ErrorException` instances.
- Wraps calls to the delegate in a try/catch block.

- Passes any caught throwables to the `createResponseFromThrowable()` factory in order to return Problem Details responses.

We recommend using custom exceptions and this middleware, as the combination allows you to focus your efforts on the positive outcome paths within your middleware.

Using it in Expressive

When using Expressive, you can then compose the `ProblemDetailsMiddleware` within route-specific pipelines, allowing you to have separate error handlers for the API parts of your application:

```
// In config/routes.php:

// Per route:
$app->get('/api/books', [
    Zend\ProblemDetails\ProblemDetailsMiddleware::class,
    Books\Action\ListBooksAction::class,
], 'books');
$app->post('/api/books', [
    Zend\ProblemDetails\ProblemDetailsMiddleware::class,
    Books\Action\CreateBookAction::class,
]);
```

Alternately, if all API endpoints have a common URI path prefix, register it as path-segregated middleware:

```
// In config/pipeline.php:

$app->pipe('/api', Zend\ProblemDetails\ProblemDetailsMiddleware::class);
```

These approaches allow you to deliver consistently structured, useful errors to your API consumers.

zend-expressive-hal

The package zendframework/zend-expressive-hal[9] provides a HAL implementation for PSR-7 applications. Currently, it allows creating PSR-7 response payloads only; we may consider parsing HAL requests at a future date, however.

zend-expressive-hal implements PSR-13 (Link Definition Interfaces)[10], and provides structures for:

- Defining relational links

- Defining HAL resources
- Composing relational links in HAL resources
- Embedding HAL resources in other HAL resources

These utilities can be used manually, without any other requirements:

```
use Zend\Expressive\Hal\HalResource;
use Zend\Expressive\Hal\Link;

$author = new HalResource($authorDataArray);
$author = $author->withLink(
    new Link('self', '/authors/' . $authorDataArray['id'])
);

$book = new HalResource($bookDataArray);
$book = $book->withLink(
    new Link('self', '/books/' . $bookDataArray['id'])
);
$book = $book->embed('authors', [$author]);
```

> Both `Link` and `HalResource` are *immutable*; as such, if you wish to make iterative changes, you will need to re-assign the original value.

These clases allow you to model the data to return in your representation, but what about returning a response based on them? To handle that, we have the `HalResponseFactory` , which will generate a response from a resource provided to it:

```
return $halResponseFactory->createResponse($request, $book);
```

> Like the `ProblemDetailsFactory` , the `HalResponseFactory` is agnostic of PSR-7 implementation, and allows you to inject a response prototype and stream factory during instantiation.
>
> Also, it, too, uses content negotiation in order to determine whether a JSON or XML response should be generated.

The above might generate the following response:

```
HTTP/1.1 200 OK
Content-Type: application/hal+json

{
  "_links": {
    "self": {"href": "/books/42"}
  },
  "id": 42
  "title": "The HitchHiker's Guide to the Galaxy",
  "_embedded": {
    "authors": [
      {
        "_links": {
          "self": {"href": "/author/12"}
        },
        "id": 12,
        "name": "Douglas Adams"
      }
    ]
  }
}
```

If your resources might be used in multiple API endpoints, you may find that creating them manually everywhere you need them is a bit of a chore!

One of the most powerful pieces of zend-expressive-hal is that it provides tools for mapping object types to how they should be represented. This is done via a *metadata map*, which maps class types to zend-hydrator extractors for the purpose of generating a representation. Additionally, we provide tools for generating link URIs based on defined routes, which allows metadata to provide dynamic link generation for generated resources.

I won't go into the architecture of how all this works, as there's a fair amount of detail. In practice, what will generally happen is:

- You'll define a *metadata map* in your application configuration, mapping your own classes to details on how to represent them.
- You'll compose a `Zend\Expressive\Hal\ResourceGenerator` (which will use a metadata map based on your configuration) and a `HalResponseFactory` in your middleware.
- You'll pass an object to the `ResourceGenerator` in order to produce a `HalResource`.
- You'll pass the generated `HalResource` to your `HalResponseFactory` to produce a response.

So, as an example, I might define the following metadata map configuration:

```
namespace Books;

use Zend\Expressive\Hal\Metadata\MetadataMap;
```

```php
use Zend\Expressive\Hal\Metadata\RouteBasedCollectionMetadata;
use Zend\Expressive\Hal\Metadata\RouteBasedResourceMetadata;
use Zend\Hydreator\ObjectProperty as ObjectPropertyHydrator;

class ConfigProvider
{
    public function __invoke() : array
    {
        return [
            'dependencies' => $this->getDependencies(),
            MetadataMap::class => $this->getMetadataMap(),
        ];
    }

    public function getDependencies() : array
    {
        return [ /* ... */ ];
    }

    public function getMetadataMap() : array
    {
        return [
            [
                '__class__' => RouteBasedResourceMetadata::class,
                'resource_class' => Author::class,
                'route' => 'author',
                'extractor' => ObjectPropertyHydrator::class,
            ],
            [
                '__class__' => RouteBasedCollectionMetadata::class,
                'collection_class' => AuthorCollection::class,
                'collection_relation' => 'authors',
                'route' => 'authors',
            ],
            [
                '__class__' => RouteBasedResourceMetadata::class,
                'resource_class' => Book::class,
                'route' => 'book',
                'extractor' => ObjectPropertyHydrator::class,
            ],
            [
                '__class__' => RouteBasedCollectionMetadata::class,
                'collection_class' => BookCollection::class,
                'collection_relation' => 'books',
                'route' => 'books',
            ],
        ];
    }
}
```

The above defines metadata for authors and books, both as individual resources as well as collections. This allows us to then embed an author as a property of a book, and have it represented as an embedded resource!

From there, we could have middleware that composes both a `ResourceGenerator` and a `HalResponseFactory` in order to produce representations:

```php
namespace Books\Action;

use Books\Repository;
use Interop\Http\ServerMiddleware\DelegateInterface;
use Interop\Http\ServerMiddleware\MiddlewareInterface;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Zend\Expressive\Hal\HalResponseFactory;
use Zend\Expressive\Hal\ResourceGenerator;

class ListBooksAction implements MiddlewareInterface
{
    private $repository;
    private $resourceGenerator;
    private $responseFactory;

    public function __construct(
        Repository $repository,
        ResourceGenerator $resourceGenerator,
        HalResponseFactory $responseFactory
    ) {
        $this->repository = $repository;
        $this->resourceGenerator = $resourceGenerator;
        $this->responseFactory = $responseFactory;
    }

    public function process(
        ServerRequestInterface $request,
        DelegateInterface $delegate
    ) : ResponseInterface {
        /** @var \Books\BookCollection $books */
        $books = $this->repository->fetchAll();

        return $this->responseFactory->createResponse(
            $request,
            $this->resourceGenerator->fromObject($books)
        );
    }
}
```

When using zend-expressive-hal to generate your responses, the majority of your middleware will look almost exactly like this!

We provide a number of other features in the package as well:

- You can define your own metadata types, and strategy classes for producing representations based on objects matching that metadata.
- You can specify custom mediatypes for your generated responses.
- You can provide your own link generation (useful if you're not using Expressive).
- You can provide your own JSON and XML renderers, if you want to vary the output for some reason (e.g., always adding specific links).

Use Anywhere!

These two packages, while part of the Zend Framework and Expressive ecosystems, can be used anywhere you use PSR-7 middleware. The Problem Details component provides a factory for producing a PSR-7 Problem Details response, and optionally middleware for automating reporting of errors. The HAL component provides only a factory for producing a PSR-7 HAL response, and a number of tools for modeling the data to return in that response.

As such, we encourage Slim, Lumen, and other PSR-7 framework users to consider using these components in your API applications to provide standard, robust, and extensible representations to your users!

For more details and examples, visit the docs for each component:

- zend-problem-details documentation: https://docs.zendframework.com/zend-problem-details
- zend-expressive-hal documentation: https://docs.zendframework.com/zend-expressive-hal

Footnotes

> 1. http://www.php-fig.org/psr/psr-7/ ↩

> 2. https://tools.ietf.org/html/rfc7807 ↩

> 3. https://tools.ietf.org/html/draft-kelly-json-hal-08 ↩

> 4. https://tools.ietf.org/html/draft-michaud-xml-hal-01 ↩

> 5. http://jsonapi.org ↩

> 6. http://amundsen.com/media-types/collection/format/ ↩

> 7. http://hyperschema.org/mediatypes/siren ↩

8

> 8. https://github.com/zendframework/zend-problem-details ↩

> 9. https://github.com/zendframework/zend-expressive-hal ↩

> 10. http://www.php-fig.org/psr/psr-13/ ↩

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/