PhlyRestfully - ZF2 module for creating RESTful JSON APIs

Release 2.1.0dev

Matthew Weier O'Phinney

July 25, 2013

CONTENTS

PhlyRestfully is a Zend Framework 2 module for implementing RESTful JSON APIs in Hypermedia Application Language (HAL). It provides a workflow for mapping persistence to resources to expose via your API.

For error reporting, it uses API-Problem.

Contents:

CHAPTER

ONE

HAL PRIMER

HAL, short for "Hypermedia Application Language", is an open specification describing a generic structure for RESTful resources. The structure it proposes readily achieves the Richardson Maturity Model's Level 3 by ensuring that each resource contains relational links, and that a standard, identifiable structure exists for embedding other resources.

Essentially, good RESTful APIs should:

- expose resources
- via HTTP, using HTTP verbs to manipulate them
- and provide canonical links to themselves, as well as link to other, related resources.

1.1 Hypermedia Type

HAL presents two hypermedia types, one for XML and one for JSON. Typically, the type is only relevant for resources returned by the API, as relational links are not usually submitted when creating, updating, or deleting resources.

The generic mediatype that HAL defines for JSON APIs is "application/hal+json".

1.2 Resources

For JSON resources, the minimum you must do is provide a "_links" property containing a "self" relational link. As an example:

```
{
    "_links": {
        "self": {
            "href": "http://example.org/api/user/matthew"
        }
    }
    "id": "matthew",
    "name": "Matthew Weier O'Phinney"
}
```

If you are including other resources embedded in the resource you are representing, you will provide an "_embedded" property, containing the named resources. Each resource will be structured as a HAL resource, and contain at least a "_links" property with a "self" relational link.

```
{
    "_links": {
        "self": {
```

```
            "href": "http://example.org/api/user/matthew"
        }
    }
    "id": "matthew",
    "name": "Matthew Weier O'Phinney",
    "_embedded": {
        "contacts": [
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mac_nibblet"
                    }
                },
                "id": "mac_nibblet",
                "name": "Antoine Hedgecock"
            },
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/spiffyjr"
                    }
                },
                "id": "spiffyjr",
                "name": "Kyle Spraggs"
            }
        ],
        "website": {
            "_links": {
                "self": {
                    "href": "http://example.org/api/locations/mwop"
                }
            },
            "id": "mwop",
            "url": "http://www.mwop.net"
        },
    }
}
```

Note that each item in the "_embedded" list can be either a resource or an array of resources. That takes us to the next topic: collections.

1.3 Collections

Collections in HAL are literally just arrays of embedded resources. A typical collection will include a "self" relational link, but also pagination links - "first", "last", "next", and "prev" are standard relations. Often APIs will also indicate the total number of resources, how many are delivered in the current payload, and potentially other metadata about the collection.

```
{
    "_links": {
        "self": {
            "href": "http://example.org/api/user?page=3"
        },
        "first": {
            "href": "http://example.org/api/user"
        },
```

```
        "prev": {
            "href": "http://example.org/api/user?page=2"
        },
        "next": {
            "href": "http://example.org/api/user?page=4"
        },
        "last": {
            "href": "http://example.org/api/user?page=133"
        }
    }
    "count": 3,
    "total": 498,
    "_embedded": {
        "users": [
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mwop"
                    }
                },
                "id": "mwop",
                "name": "Matthew Weier O'Phinney"
            },
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mac_nibblet"
                    }
                },
                "id": "mac_nibblet",
                "name": "Antoine Hedgecock"
            },
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/spiffyjr"
                    }
                },
                "id": "spiffyjr",
                "name": "Kyle Spraggs"
            }
        ]
    }
}
```


The various relational links for the collection make it trivial to traverse the API to get a full list of resources in the collection. You can easily determine what page you are on, and what the next page should be (and if you are on the last page).

Each item in the collection is a resource, and contains a link to itself, so you can get the full resource, but also know its canonical location. Often, you may not embed the full resource in a collection – just the bits that are relevant when doing a quick list. As such, having the link to the individual resource allows you to get the full details later if desired.

1.4 Interacting with HAL

Interacting with HAL is usually quite straight-forward:

- Make a request, using the Accept header with a value of `application/json` or `application/hal+json` (the latter really isn't necessary, though).
- If ``POST``ing, ``PUT``ting, ``PATCH``ing, or ``DELETE``ing a resource, you will usually use a Content-Type header of either ``application/json``, or some vendor-specific mediatype you define for your API; this mediatype would be used to describe the particular structure of your resources _without_ any HAL "_links". Any "_embedded" resources will typically be described as properties of the resource, and point to the mediatype relevant to the embedded resource.
- The API will respond with a mediatype of `application/hal+json`.

When creating or updating a resource (or collection), you will submit the object, without relational links; the API is responsible for assigning the links. If we consider the embedded resources example from above, I would create it like this:

```
POST /api/user
Accept: application/json
Content-Type: application/vnd.example.user+json

{
    "id": "matthew",
    "name": "Matthew Weier O'Phinney",
    "contacts": [
        {
            "id": "mac_nibblet",
        },
        {
            "id": "spiffyjr",
        }
    ],
    "website": {
        "id": "mwop",
    }
}
```

The response would look like this:

```
HTTP/1.1 201 Created
Content-Type: application/hal+json
Location: http://example.org/api/user/matthew

{
    "_links": {
        "self": {
            "href": "http://example.org/api/user/matthew"
        }
    }
    "id": "matthew",
    "name": "Matthew Weier O'Phinney",
    "_embedded": {
        "contacts": [
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mac_nibblet"
                    }
                },
                "id": "mac_nibblet",
                "name": "Antoine Hedgecock"
            },
```

```
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/spiffyjr"
                    }
                },
                "id": "spiffyjr",
                "name": "Kyle Spraggs"
            }
        ],
        "website": {
            "_links": {
                "self": {
                    "href": "http://example.org/api/locations/mwop"
                }
            },
            "id": "mwop",
            "url": "http://www.mwop.net"
        },
    }
}
```

`PUT` and `PATCH` operate similarly.

CHAPTER

TWO

ERROR REPORTING

HAL does a great job of defining a generic mediatype for resources with relational links. However, how do you go about reporting errors? HAL is silent on the issue.

REST advocates indicate that HTTP response status codes should be used, but little has been done to standardize on the response format.

For JSON APIs, though, two formats are starting to achieve large adoption: `vnd.error` and `API-Problem`. In PhlyRestfully, I have provided support for returning `Api-Problem` payloads.

2.1 API-Problem

This mediatype, `application/api-problem+json` is via the IETF, and actually also includes an XML variant. The structure includes the following properties:

- **describedBy**: a URL to a document describing the error condition (required)
- **title**: a brief title for the error condition (required)
- **httpStatus**: the HTTP status code for the current request (optional)
- **detail**: error details specific to this request (optional)
- **supportId**: a URL to the specific problem occurrence (e.g., to a log message) (optional)

As an example payload:

```
HTTP/1.1 500 Internal Error
Content-Type: application/api-problem+json

{
    "describedBy": "http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
    "detail": "Status failed validation",
    "httpStatus": 500,
    "title": "Internal Server Error"
}
```

The specification allows a large amount of flexibility – you can have your own custom error types, so long as you have a description of them to link to. You can provide as little or as much detail as you want, and even decide what information to expose based on environment.

2.2 PhlyRestfully Choices

The specification indicates that every error needs to include a "describedBy" field, pointing to a URI with more information on the problem type. Often, when you start a project, you may not want to worry about this up front – the HTTP status code may be enough to begin. As such, PhlyRestfully assumes that if you do not provide a "describedBy" field, it will link to the URI describing the HTTP status codes.

CHAPTER

THREE

WHITELISTING HTTP METHODS

If your API is to adhere to the Richardson Maturity Model's level 2 or higher, you will be using HTTP verbs to interact with it: `GET`, `POST`, `PUT`, `DELETE`, and `PATCH` being the most common. However, based on the resource and whether or not the end point is a collection, you may want to allow different HTTP methods. How can you do that? and how do you enforce it?

HTTP provides functionality around this topic via another HTTP method, `OPTIONS`, and a related HTTP response header, `Allow`.

Calls to `OPTIONS` are non-cacheable, and may provide a response body if desired. They _should_ emit an `Allow` header, however, detailing which HTTP request methods are allowed on the current URI.

Consider the following request:

```
OPTIONS /api/user
Host: example.org
```

with its response:

```
HTTP/1.1 200 OK
Allow: GET, POST
```

This tells us that for the URI `/api/user`, you may emit either a `GET` or `POST` request.

What happens if a malicious user tries something else? You should respond with a "405 Not Allowed" status, and indicate what _is_ allowed:

```
HTTP/1.1 405 Not Allowed
Allow: GET, POST
```

PhlyRestfully bakes this into its `PhlyRestfully\ResourceController` implementation, allowing you to specify via configuration which methods are allowed both for collections and individual resources handled by the controller.

CHAPTER

FOUR

PHLYRESTFULLY BASICS

PhlyRestfully allows you to create RESTful JSON APIs that adhere to *Hypermedia Application Language*. For error handling, it uses *API-Problem*.

The pieces you need to implement, work with, or understand are:

- Writing event listeners for the various `PhlyRestfully\Resource` events, which will be used to either persist resources or fetch resources from persistence.
- Writing routes for your resources, and associating them with resources and/or `PhlyRestfully\ResourceController`.
- Writing metadata describing your resources, including what routes to associate with them.

All API calls are handled by `PhlyRestfully\ResourceController`, which in turn composes a `PhlyRestfully\Resource` object and calls methods on it. The various methods of the controller will return either `PhlyRestfully\ApiProblem` results on error conditions, or, on success, a `PhlyRestfully\HalResource` or `PhlyRestfully\HalCollection` instance; these are then composed into a `PhlyRestfully\View\RestfulJsonModel`.

If the MVC detects a `PhlyRestfully\View\RestfulJsonModel` during rendering, it will select `PhlyRestfully\View\RestfulJsonRenderer`. This, with the help of the `PhlyRestfully\Plugin\HalLinks` plugin, will generate an appropriate payload based on the object composed, and ensure the appropriate Content-Type header is used.

If a `PhlyRestfully\HalCollection` is detected, and the renderer determines that it composes a `Zend\Paginator\Paginator` instance, the `HalLinks` plugin will also generate pagination relational links to render in the payload.

CHAPTER

FIVE

RESOURCES

In order to perist resources or retrieve resources to represent, PhlyRestfully uses a `PhlyRestfully\Resource` instance. This class simply triggers an event based on the operation requested; you, as a developer, provide and attach listeners to those events to do the actual work.

`PhlyRestfully\Resource` defines the following events, with the following event parameters:

Event name	Parameters
create	
update	data
replaceList	data
patch	data
delete	
deleteList	
fetch	
fetchAll	

Event listeners receive an instance of `PhlyRestfully\ResourceEvent`, which also composes the route matches and query parameters from the request. You may retrieve them from the event instance using the following methods:

- `getQueryParams()` (returns a `Zend\Stdlib\Parameters` instance)
- `getRouteMatch()` (returns a `Zend\Mvc\Router\RouteMatch` instance)
- `getQueryParam($name, $default = null)`
- `getRouteParam($name, $default = null)`

Within your listeners, you have the option of throwing an exception in order to raise an `ApiProblem` response. The following maps events to the special exceptions you can raise; all exceptions are in the `PhlyRestfully\Exception` namespace, except where globally qualified:

Event name Parameters
create `CreationException`
update `UpdateException`
replaceList `UpdateException`
patch `PatchException`
delete `\Exception`
deleteList `\Exception`
fetch `\Exception`
fetchAll `\Exception`

Additionally, if you throw any exception implementing `PhlyRestfully\Exception\ProblemExceptionInterface`, it can be used to seed an `ApiProblem` instance with the appropriate information. Such an exception needs to define the following methods:

- **getAdditionalDetails**(), which should return a string or array.
- **getDescribedBy**(), which should return a URI for the "describedBy" field.
- **getTitle**(), which should return a string for the "title" field.

The exception code and message will be used for the "httpStatus" and "detail", respectively.

The `CreationException`, `UpdateException`, and `PatchException` types all inherit from `DomainException`, which implements the `ProblemExceptionInterface`.

As a quick example, let's look at two listeners, one that listens on the `create` event, and another on the `fetch` event, in order to see how we might handle them.

```
// listener on "create"
function ($e) {
    $data = $e->getParam('data');

    // Assume an ActiveRecord-like pattern here for simplicity
    $user = User::factory($data);
    if (!$user->isValid()) {
        $ex = new CreationException('New user failed validation', 400);
        $ex->setAdditionalDetails($user->getMessages());
        $ex->setDescribedBy('http://example.org/api/errors/user-validation');
        $ex->setTitle('Validation error');
        throw $ex;
```

```
    }

    $user->persist();
    return $user;
}

// listener on "fetch"
function ($e) {
    $id = $e->getParam('id');

    // Assume an ActiveRecord-like pattern here for simplicity
    $user = User::fetch($id);
    if (!$user) {
        $ex = new DomainException('User not found', 404);
        $ex->setDescibedBy('http://example.org/api/errors/user-not-found');
        $ex->setTitle('User not found');
        throw $ex;
    }

    return $user;
}
```

Typically, you will create a `Zend\EventManager\ListenerAggregateInterface` implementation that will contain all of your listeners, so that you can also compose in other classes such as data mappers, a service layer, etc. Read about listener aggregates in the ZF2 documentation if you are unfamiliar with them.

In a later section, I will show you how to wire your listener aggregate to a resource and resource controller.

CHAPTER

SIX

RESOURCECONTROLLERS

While the `Resource` hands off work to your domain logic, `PhlyRestfully\ResourceController` mediates between the incoming request and the `Resource`, as well as ensures an appropriate response payload is created and returned.

For the majority of cases, you should be able to use the `ResourceController` unmodified; you will only need to provide it with a `Resource` instance and some configuration detailing what Content-Types to respond to, what constitutes an acceptable Accept header, and what HTTP methods are valid for both collections and individual resources.

A common factory for a `ResourceController` instance might look like the following:

```
return array(
    'controllers' => array(
        'PasteController' => function ($controllers) {
            $services    = $controllers->getServiceLocator();

            $persistence = $services->get('PastePersistenceListener');
            $events      = $services->get('EventManager');
            $events->setIdentifiers('PasteResource');
            $events->attach($persistence);

            $resource    = new PhlyRestfully\Resource();
            $resource->setEventManager($events);

            $controller = new PhlyRestfully\ResourceController('PasteController');
            $controller->setResource($resource);
            $controller->setRoute('paste/api');
            $controller->setCollectionName('pastes');
            $controller->setPageSize(30);
            $controller->setCollectionHttpOptions(array(
                'GET',
                'POST',
            ));
            $controller->setResourceHttpOptions(array(
                'GET',
            ));

            return $controller;
        },
    ),
);
```

Essentially, three steps are taken:

- A listener is pulled from the service manager, and injected into a new event manager instance.

- A `Resource` instance is created, and injected with the event manager instance.
- A `ResourceController` instance is created, and injected with the `Resource` instance and some configuration.

Considering that most `ResourceController` instances follow the same pattern, PhlyRestfully provides an abstract factory for controllers that does the work for you. To use it, you will provide a `resources` subkey in your `phlyrestfully` configuration, with controller name/configuration pairs. As an example:

```
// In a module's configuration, or the autoloadable configuration of your
// application:
return array(
    'phlyrestfully' => array(
        'resources' => array(
            // Key is the service name for the controller; value is
            // configuration
            'MyApi\Controller\Contacts' => array(
                // Name of the controller class to use, if other than
                // PhlyRestfully\ResourceController. Must extend
                // PhlyRestfully\ResourceController, however, to be valid.
                // (OPTIONAL)
                'controller_class' => 'PhlyRestfully\ResourceController',

                // Event identifier for the resource controller. By default,
                // the resource name is used; you can use a different
                // identifier via this key.
                // (OPTIONAL)
                'identifier' => 'Contacts',

                // Name of the service locator key OR the fully qualified
                // class name of the resource listener (latter works only if
                // the class has no required arguments in the constructor).
                // (REQUIRED)
                'listener'   => 'MyApi\Resource\Contacts',

                // Event identifiers for the composed resource. By default,
                // the class name of the listener is used; you can add another
                // identifier, or an array of identifiers, via this key.
                // (OPTIONAL)
                'resource_identifiers' => array('ContactsResource'),

                // Accept criteria (which accept headers will be allowed)
                // (OPTIONAL)
                'accept_criteria' => array(
                    'PhlyRestfully\View\RestfulJsonModel' => array(
                        'application/json',
                        'text/json',
                    ),
                ),

                // HTTP options for resource collections
                // (OPTIONAL)
                'collection_http_options' => array('get', 'post'),

                // Collection name (OPTIONAL)
                'collection_name' => 'contacts',

                // Query parameter or array of query parameters that should be
                // injected into collection links if discovered in the request.
```

```
                // By default, only the "page" query parameter will be present.
                // (OPTIONAL)
                'collection_query_whitelist' => 'sort',

                // Content types to respond to
                // (OPTIONAL)
                'content_type' => array(
                    ResourceController::CONTENT_TYPE_JSON => array(
                        'application/json',
                        'application/hal+json',
                        'text/json',
                    ),
                ),

                // If a custom identifier_name is used
                // (OPTIONAL)
                'identifier_name'  => 'contact_id',

                // Number of items to return per page of a collection
                // (OPTIONAL)
                'page_size'  => 30,

                // Query string parameter that will indicate number of items
                // per page of results. If this is set, and the parameter is
                // passed, it will be used in favor of the page_size.
                // Leaving it unset will disable the ability of the client
                // to set the page size via query string parameter.
                // (OPTIONAL)
                'page_size_param' => 'page_size',

                // HTTP options for individual resources
                // (OPTIONAL)
                'resource_http_options'   => array('get', 'patch', 'put', 'delete'),

                // name of the route associated with this resource
                // (REQUIRED)
                'route_name' => 'api/contacts',
            ),
        ),
    ),
);
```

The options defined above cover every available configuration option of the `ResourceController`, and ensure that your primary listener for the `Resource` is attached. Additionally, it ensures that both your `Resource` and `ResourceController` have defined identifiers for their composed event manager instances, allowing you to attach shared event listeners - which can be useful for implementing logging, caching, authentication and authorization checks, etc.

Note that the above configuration assumes that you are defining a `Zend\EventManager\ListenerAggregateInterface` implementation to attach to the `Resource`. This is a good practice anyways, as it keeps the logic encapsulated, and allows you to have stateful listeners – which is particularly useful as most often you will consume a mapper or similar within your listeners in order to persist resources or fetch resources from persistence.

CHAPTER

SEVEN

EXAMPLE

The following is an example detailing a service that allows creating new resources and fetching existing resources only. It could be expanded to allow updating, patching, and deletion, but the basic premise stays the same.

First, I'll define an interface for persistence. I'm doing this in order to focus on the pieces related to the API; how you actually persist your data is completely up to you.

```
1  namespace Paste;
2
3  interface PersistenceInterface
4  {
5      public function save(array $data);
6      public function fetch($id);
7      public function fetchAll();
8  }
```

Next, I'll create a resource listener. This example assumes you are using Zend Framework 2.2.0 or above, which includes the `AbstractListenerAggregate`; if you are using a previous version, you will need to manually implement the `ListenerAggregateInterface` and its `detach()` method.

```
1  namespace Paste;
2
3  use PhlyRestfully\Exception\CreationException;
4  use PhlyRestfully\Exception\DomainException;
5  use PhlyRestfully\ResourceEvent;
6  use Zend\EventManager\AbstractListenerAggregate;
7  use Zend\EventManager\EventManagerInterface;
8
9  class PasteResourceListener extends AbstractListenerAggregate
10 {
11     protected $persistence;
12
13     public function __construct(PersistenceInterface $persistence)
14     {
15         $this->persistence = $persistence;
16     }
17
18     public function attach(EventManagerInterface $events)
19     {
20         $this->listeners[] = $events->attach('create', array($this, 'onCreate'));
21         $this->listeners[] = $events->attach('fetch', array($this, 'onFetch'));
22         $this->listeners[] = $events->attach('fetchAll', array($this, 'onFetchAll'));
23     }
24
```

```
25      public function onCreate(ResourceEvent $e)
26      {
27          $data  = $e->getParam('data');
28          $paste = $this->persistence->save($data);
29          if (!$paste) {
30              throw new CreationException();
31          }
32          return $paste;
33      }
34
35      public function onFetch(ResourceEvent $e)
36      {
37          $id = $e->getParam('id');
38          $paste = $this->persistence->fetch($id);
39          if (!$paste) {
40              throw new DomainException('Paste not found', 404);
41          }
42          return $paste;
43      }
44
45      public function onFetchAll(ResourceEvent $e)
46      {
47          return $this->persistence->fetchAll();
48      }
49  }
```

The job of the listeners is to pull arguments from the passed event instance, and then work with the persistence storage. Based on what is returned we either throw an exception with appropriate messages and/or codes, or we return a result.

Now that we have a resource listener, we can begin integrating it into our application.

For the purposes of our example, we'll assume:

- The persistence engine is returning arrays (or arrays of arrays, when it comes to `fetchAll()`.
- The identifier field in each array is simply "id".

First, let's create a route. In our module's configuration file, usually `config/module.config.php`, we'd add the following routing definitions:

```
1   'router' => array('routes' => array(
2       'paste' => array(
3           'type' => 'Literal',
4           'options' => array(
5               'route' => '/paste',
6               'controller' => 'Paste\PasteController', // for the web UI
7           ),
8           'may_terminate' => true,
9           'child_routes' => array(
10              'api' => array(
11                  'type' => 'Segment',
12                  'options' => array(
13                      'route'      => '/api/pastes[/:id]',
14                      'controller' => 'Paste\ApiController',
15                  ),
16              ),
17          ),
18      ),
19  )),
```

I defined a top-level route for the namespace, which will likely be accessible via a web UI, and will have a different controller. For the purposes of this example, we'll ignore that for now. The import route is `paste/api`, which is our RESTful endpoint.

Next, let's define the controller configuration. Again, inside our module configuration, we'll add configuration, this time under the `phlyrestfully` key and its `resources` subkey.

```
'phlyrestfully' => array(
    'resources' => array(
        'Paste\ApiController' => array(
            'identifier'              => 'Pastes',
            'listener'                => 'Paste\PasteResourceListener',
            'resource_identifiers'    => array('PasteResource'),
            'collection_http_options' => array('get', 'post'),
            'collection_name'         => 'pastes',
            'page_size'               => 10,
            'resource_http_options'   => array('get'),
            'route_name'              => 'paste/api',
        ),
    ),
),
```

Notie that the configuration is a subset of all configuration at this point; we're only defining the options needed for our particular resource.

Now, how can we get our `PasteResourceListener` instance? Remember, it requires a `PersistenceInterface` instance to the constructor. Let's add a factory inside our `Module` class. The full module class is presented here.

```
namespace Paste;

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

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

    public function getServiceConfig()
    {
        return array('factories' => array(
            'Paste\PasteResourceListener' => function ($services) {
                $persistence = $services->get('Paste\PersistenceInterface');
                return new PasteResourceListener($persistence);
            },
        ));
```

```
29         }
30     }
```

Note: I lied: I'm not giving the full configuration. The reason is that I'm not defining the actual persistence implementation in the example. If you continue with the example, you would need to define it, and assign a factory to the service name `Paste\PersistenceInterface`.

At this point, we're done! Register your module with the application configuration (usually `config/application.config.php`), and you should immediately be able to access the API.

Note: When hitting the API, make sure you send an Accept header with either the content type `application/json`, `application/hal+json`, or `text/json`; otherwise, it will try to deliver HTML to you, and, unless you have defined view scripts accordingly, you will see errors.

CHAPTER

EIGHT

THE RESOURCEEVENT

When `PhlyRestfully\Resource` triggers events, it passes a custom event type, `PhlyRestfully\ResourceEvent`. This custom event contains several additional methods which allow you to access route match and query parameters, which are often useful when working with child routes or wanting to provide sorting, filtering, or other actions on collections.

The available methods are:

- `getRouteMatch()`, which returns the `Zend\Mvc\Router\RouteMatch` instance that indicates the currently active route in the MVC, and contains any parameters matched during routing.
- `getRouteParam($name, $default = null)` allows you to retrieve a single route match parameter.
- `getQueryParams()` returns the collection of query parameters from the current request.
- `getQueryParam($name, $default = null)` allows you to retrieve a single query parameter.

The `ResourceEvent` is created internal to the `Resource`, and cloned for each event triggered. If you would like to pass additional parameters, the `Resource` object allows this, via its `setEventParams()` method, which accepts an associative array of named parameters.

As an example, if you were handling authentication via a custom HTTP header, you could pull this in a listener, and pass it to the resource as follows; the following is the body of a theoretical `onBootstrap()` method of your `Module` class.

```
$target       = $e->getTarget();
$events       = $target->getEventManager();
$sharedEvents = $events->getSharedManager();
$sharedEvents->attach('Paste\ApiController', 'create', function ($e) {
    $request  = $e->getRequest();
    $headers  = $request->getHeaders();

    if (!$headers->has('X-Paste-Authentication')) {
        return;
    }

    $auth     = $headers->get('X-Paste-Authentication')->getFieldValue();
    $target   = $e->getTarget();
    $resource = $target->getResource();

    $resource->setEventParams(array(
        'auth' => $auth,
    ));
}, 100);
```

The above grabs the header, if it exists, and passes it into the resource as an event parameter. Later, in a listener, you can grab it:

```
$auth = $e->getParam('auth', false);
```

CHAPTER

NINE

CONTROLLER EVENTS

Each of the various REST endpoint methods - `create()`, `delete()`, `deleteList()`, `get()`, `getList()`, `patch()`, `update()`, and `replaceList()` - trigger both a `{methodname}.pre` and a `{methodname}.post` event.

The "pre" event is executed after validating arguments, and will receive any arguments passed to the method; the "post" event occurs right before returning from the method, and receives the same arguments, plus the resource or collection, if applicable.

These methods are useful in the following scenarios:

- Specifying custom HAL links
- Aggregating additional request parameters to pass to the resource object

As an example, if you wanted to add a "describedby" HAL link to every resource or collection returned, you could do the following:

```
// Methods we're interested in
$methods = array(
    'create.post',
    'get.post',
    'getList.post',
);

// Assuming $sharedEvents is a ZF2 SharedEventManager instance
$sharedEvents->attach('Paste\ApiController', $methods, function ($e) {
    $resource = $e->getParam('resource', false);
    if (!$resource) {
        $resource = $e->getParam('collection', false);
    }

    if (!$resource instanceof \PhlyRestfully\LinkCollectionAwareInterface) {
        return;
    }

    $link = new \PhlyRestfully\Link('describedby');
    $link->setRoute('paste/api/docs');
    $resource->getLinks()->add($link);
});
```

CHAPTER

TEN

ADVANCED ROUTING

The recommended route for a resource is a `Zend\Mvc\Router\Http\Segment` route, with an identifier:

```
'route' => '/resource[/:id]'
```

This works great for standalone resources, but poses a problem for hierarchical resources. As an example, if you had a "users" resource, but then had "addresses" that were managed as part of the user, the following route definition poses a problem:

```
'users' => array(
    'type' => 'Segment',
    'options' => array(
        'route' => '/users[/:id]',
        'controller' => 'UserResourceController',
    ),
    'may_terminate' => true,
    'child_routes' => array(
        'addresses' => array(
            'type' => 'Segment',
            'options' => array(
                'route' => '/addresses[/:id]',
                'controller' => 'UserAddressResourceController',
            ),
        ),
    ),
),
```

Spot the problem? Both the parent and child have an "id" segment, which means there is a conflict. Let's refactor this a bit:

```
'users' => array(
    'type' => 'Segment',
    'options' => array(
        'route' => '/users[/:user_id]',
        'controller' => 'UserResourceController',
    ),
    'may_terminate' => true,
    'child_routes' => array(
        'type' => 'Segment',
        'options' => array(
            'route' => '/addresses[/:address_id]',
            'controller' => 'UserAddressResourceController',
        ),
```

```
    ),
),
```

Now we have a new problem, or rather, two new problems: by default, the `ResourceController` uses "id" as the identifier, and this same identifier name is used to generate URIs. How can we change that?

First, the `ResourceController` allows you to define the identifier name for the specific resource being exposed. You can do this via the `setIdentifierName()` method, but more commonly, you'll handle it via the `identifier_name` configuration parameter:

```
'phlyrestfully' => array(
    'resources' => array(
        'UserResourceController' => array(
            // ...
            'identifier_name' => 'user_id',
            // ...
        ),
        'UserAddressResourceController' => array(
            // ...
            'identifier_name' => 'address_id',
            // ...
        ),
    ),
),
```

If you are rendering child resources as part of a resource, however, you need to hint to the renderer about where to look for an identifier.

There are several mechanisms for this: the `getIdFromResource` and `createLink` events of the `PhlyRestfully\Plugin\HalLinks` plugin; or *a metadata map*.

The `HalLinks` events are as followed, and triggered by the methods specified:

Event name Parameters		Method triggering event
createLink	`createLink`	• route * • id • resource • params *
getIdFromResource	`getIdFromResource`	• resource *

Let's dive into each of the specific events.

Note: In general, you shouldn't need to tie into the events listed on this page very often. The recommended way to customize URL generation for resources is to instead use *a metadata map*.

10.1 createLink event

The `createLink` method is currently called only from `PhlyRestfully\ResourceController::create()`, and is used to generate the `Location` header. Essentially, what it does is call the `url()` helper with the passed

route, and the `serverUrl()` helper with that result to generate a fully-qualified URL.

If passed a resource identifier and resource, you can attach to the event the method triggers in order to modifiy the route parameters and/or options when generating the link.

Consider the following scenario: you need to specify an alternate routing parameter to use for the identifier, and you want to use the "user" associated with the resource as a route parameter. Finally, you want to change the route used to generate this particular URI.

The following will do that:

```
$request = $services->get('Request');
$sharedEvents->attach('PhlyRestfully\Plugin\HalLinks', 'createLink', function ($e) use ($request) {
    $resource = $e->getParam('resource');
    if (!$resource instanceof Paste) {
        // only react for a specific type of resource
        return;
    }

    // The parameters here are an ArrayObject, which means we can simply set
    // the values on it, and the method calling us will use those.
    $params = $e->getParams();

    $params['route'] = 'paste/api/by-user';

    $id   = $e->getParam('id');
    $user = $resource->getUser();
    $params['params']['paste_id'] = $id;
    $params['params']['user_id']  = $user->getId();
}, 100);
```

The above listener will change the route used to "paste/api/by-user", and ensure that the route parameters "paste_id" and "user_id" are set based on the resource provided.

The above will be called with `create` is successful. Additionally, you can use the `HalLinks` plugin from other listeners or your view layer, and call the `createLink()` method manually – which will also trigger any listeners.

10.2 getIdFromResource event

The `getIdFromResource` event is only indirectly related to routing. Its purpose is to retrieve the identifier for a given resource so that a "self" relational link may be generated; that is its sole purpose.

The event receives exactly one argument, the resource for which the identifier is needed. A default listener is attached, at priority 1, that uses the following algorithm:

- If the resource is an array, and an "id" key exists, it returns that value.
- If the resource is an object and has a public "id" property, it returns that value.
- If the resource is an object, and has a public `getId()` method, it returns the value returned by that method.

In all other cases, it returns a boolean `false`, which generally results in an exception or other error.

This is where you, the developer come in: you can write a listener for this event in order to return the identifier yourself.

As an example, let's consider the original example, where we have "user" and "address" resources. If these are of specific types, we could write listeners like the following:

```php
$sharedEvents->attach('PhlyRestfully\Plugin\HalLinks', 'getIdFromResource', function ($e) {
    $resource = $e->getParam('resource');
    if (!$resource instanceof User) {
        return;
    }
    return $resource->user_id;
}, 100);

$sharedEvents->attach('PhlyRestfully\Plugin\HalLinks', 'getIdFromResource', function ($e) {
    $resource = $e->getParam('resource');
    if (!$resource instanceof UserAddress) {
        return;
    }
    return $resource->address_id;
}, 100);
```

Since writing listeners like these gets old quickly, I recommend using *a metadata map* instead.

CHAPTER

ELEVEN

ADVANCED RENDERING

What if you're not returning an array as a resource from your persistence layer? Somehow, you have to be able to transform it to an associative array so that it can be rendered to JSON easily. There are a variety of ways to do this, obviously; the easiest would be to make your resource implement `JsonSerializable`. If that's not an option, though, what other approaches do you have?

In this section, we'll explore one specific solution that is the most explicit of those available: the `renderCollection.resource` event of the `HalLinks` plugin.

11.1 HalLinks overview

`PhlyRestfully\Plugin\HalLinks` acts as both a controller plugin as well as a view helper. In most cases, you likely will not interact directly with it. However, it does expose a few pieces of functionality that may be of interest:

- The `createLink()` method, which is handy for creating fully-qualified URLs (i.e., contains schema, hostname, and port, in addition to path). This was detailed *in the previous section*.
- The `getIdFromResource` event (also detailed *in the previous section*).
- The `renderCollection.resource` event.

If in a controller, or interacting with a controller instance, you can access it via the controller's `plugin()` method:

```
$halLinks = $controller->plugin('HalLinks');
```

For the purposes of this chapter, we'll look specifically at the `renderCollection.resource` event, as it allows you, the developer, to fully customize how you extract your resource to an array.

11.2 The renderCollection.resource event

This method is triggered as part of the `renderCollection()` method, once for each resource in the collection. It receives the following parameters:

- **collection**, the current collection being iterated
- **resource**, the current resource in the iteration
- **route**, the resource route defined for the collection; usually, this is the same route as provided to the controller.
- **routeParams**, any route params defined for resources within the collection; usually, this is empty.
- **routeOptions**, any route options defined for resources within the collection; usually, this is empty.

Let's consider the following scenario: we're rendering something like a public status timeline, but the individual status resources in our timeline belong to another route. Additionally, we want to show a subset of information for each individual status when in the public timeline; we don't need the full status resource.

We'd define a listener:

```
$sharedEvents->attach('PhlyRestfully\Plugin\HalLinks', 'renderCollection.resource', function ($e) {
    $collection = $e->getParam('collection');
    if (!$collection instanceof PublicTimeline) {
        // nothing to do here
        return;
    }

    $resource = $e->getParam('resource');
    if (!$resource instanceof Status) {
        // nothing to do here
        return;
    }

    $return = array(
        'id'        => $resource->getId(),
        'user'      => $resource->getUser(),
        'timestamp' => $resource->getTimestamp(),
    );

    // Parameters are stored as an ArrayObject, allowing us to change them
    // in situ
    $params = $e->getParams();
    $params['resource']    = $return;
    $params['route']       = 'api/status/by-user';
    $params['routeParams'] = array(
        'id'   => $resource->getId(),
        'user' => $resource->getUser(),
    );
}, 100);
```

The above extracts three specific fields of the `Status` object and creates an array representation for them. Additionally, it changes the route used, and sets some route parameters. This information will be used when generating a "self" relational link for the resource, and the newly created array will be used when creating the representation for the resource itself.

This approach gives us maximum customization during the rendering process, but comes at the cost of added boiler plate code. As per the section on routing, I recommend using *a metadata map* unless you need to dynamically determine route parameters or filter the resource before rendering. Additionally, in many cases *hydrators* (the subject of the next section) are more than sufficient for the purpose of creating an array representation of your resource.

CHAPTER

TWELVE

HYDRATORS

`Zend\Stdlib\Hydrator` offers a general-purpose solution for mapping arrays to objects (hydration) and objects to arrays (extraction). In PhlyRestfully, hydrators are used during rendering for this second operation, extraction, so that resources may be represented via JSON.

Within PhlyRestfully, `PhlyRestfully\View\JsonRenderer` delegates to `PhlyRestfully\Plugin\HalLinks` in order to return a representation of a resource or collection. This was done to allow you, the user, to override how rendering is accomplished if desired; you can extend the `HalLinks` plugin and register your own version as a controller plugin and view helper.

Since `HalLinks` handles the conversion, it also acts as a registry for mapping classes to the hydrators responsible for extracting them.

12.1 Manually working with the hydrator map

If you want to programmatically associate classes with their hydrators, you can grab an instance of the `HalLinks` plugin in several ways:

- via the view helper manager
- via the controller plugin manager

To extract it from the view helper manager:

```
// Assuming we're in a module's onBootstrap method or other listener on an
// application event:
$app = $e->getApplication();
$services = $app->getServiceManager();
$helpers  = $services->get('ViewHelperManager');
$halLinks = $helpers->get('HalLinks');
```

Similarly, you can grab it from the controller plugin manager:

```
// Assuming we're in a module's onBootstrap method or other listener on an
// application event:
$app = $e->getApplication();
$services = $app->getServiceManager();
$plugins  = $services->get('ControllerPluginManager');
$halLinks = $plugins->get('HalLinks');
```

Alternately, if listening on a controller event, pull it from the controller's `plugin()` method:

```
$controller = $e->getTarget();
$halLinks   = $controller->plugin('HalLinks');
```

Once you have the plugin, you can register class/hydrator pairs using the `addHydrator()` method:

```
// Instantiate the hydrator instance directly:
$hydrator = new \Zend\Stdlib\Hydrator\ClassMethods();

// Or pull it from the HydratorManager:
$hydrators = $services->get('HydratorManager');
$hydrator  = $hydrators->get('ClassMethods');

// Then register it:
$halLinks->addHydrator('Paste\PasteResource', $hydrator);

// More succintly, since HalLinks composes the HydratorManager by default,
// you can use te short name of the hydrator service:
$halLinks->addHydrator('Paste\PasteResource', 'ClassMethods');
```

All done!

You can also specify a default hydrator to use, if `HalLinks` can't find the resource class in the map:

```
$halLinks->setDefaultHydrator($hydrator);
```

However, it's a lot of boiler plate code. There is a simpler way: configuration.

12.2 Configuration-driven hydrator maps

You can specify hydrators to use with the objects you return from your resources via configuration, and you can specify both a map of class/hydrator service pairs as well as a default hydrator to use as a fallback. As an example, consider the following *config/autoload/phlyrestfully.global.php* file:

```
return array(
    'phlyrestfully' => array(
        'renderer' => array(
            'default_hydrator' => 'ArraySerializable',
            'hydrators' => array(
                'My\Resources\Foo' => 'ObjectProperty',
                'My\Resources\Bar' => 'Reflection',
            ),
        ),
    ),
);
```

The above specifies `Zend\Stdlib\Hydrator\ArraySerializable` as the default hydrator, and maps the `ObjectProperty` hydrator to the `Foo` resource, and the `Reflection` hydrator to the `Bar` resource. Note that the short name for the hydrator is used; `HalLinks` composes the `HydratorManager` service by default, and pulls hydrators from there if provided by service name.

This is a cheap and easy way to ensure that you can extract your resources to arrays to be used as JSON representations.

CHAPTER

THIRTEEN

COLLECTIONS AND PAGINATION

In most use cases, you'll not want to return a collection containing every resource in that collection; this will quickly get untenable as the number of resources in that collection grows. This means you'll want to paginate your collections somehow, returning a limited set of resources at a time, delimited by some offset in the URI (usually via query string).

Additionally, to follow the Richardson Maturity Model properly, you will likely want to include relational links indicating the next and previous pages (if any), and likely the first and last as well (so that those traversing the collection know when to stop).

This gets tedious very quickly.

Fortunately, PhlyRestfully can automate the process for you, assuming you are willing to use `Zend\Paginator` to help do some of the heavy lifting.

13.1 Paginators

ZendPaginator is a general purpose component for paginating collections of data. It requires only that you specify the number of items per page of data, and the current page.

The integration within PhlyRestfully for `Zend\Paginator` uses a "page" query string variable to indicate the current page. You set the page size during configuration:

```
return array(
    'phlyrestfully' => array(
        'resources' => array(
            'Paste\ApiController' => array(
                // ...
                'page_size' => 10, // items per page of data
                // ...
            ),
        ),
    ),
);
```

All you need to do, then, is return a `Zend\Paginator\Paginator` instance from your resource listener (or an extension of that class), and PhlyRestfully will then generate appropriate relational links.

For example, if we consider the *walkthrough example*, if our `onFetchAll()` method were to return a `Paginator` instance, the collection included 3000 records, we'd set the page size to 10, and the request indicated page 17, our response would include the following links:

```
{
    "_links": {
        "self": {
            "href": "http://example.org/api/paste?page=17
        },
        "prev": {
            "href": "http://example.org/api/paste?page=16
        },
        "next": {
            "href": "http://example.org/api/paste?page=18
        },
        "first": {
            "href": "http://example.org/api/paste
        },
        "last": {
            "href": "http://example.org/api/paste?page=300
        }
    },
    // ...
}
```

Again, this functionality is built-in to PhlyRestfully; all you need to do is return a `Paginator` instance, and set the `page_size` configuration for your resource controller.

13.2 Manual collection links

If you do not want to use a `Paginator` for whatever reason, you can always listen on one of the controller events that returns a collection, and manipulate the returned `HalCollection` from there. The events of interest are:

- **getList.post**
- **replaceList.post**

In each case, you can retrieve the `HalCollection` instance via the `collection` parameter:

```
$collection = $e->getParam('collection');
```

From there, you will need to retrieve the collection's `LinkCollection`, via the `getLinks()` method, and manually inject `Link` instances. The following creates a "prev" relational link based on some calculated offset.

```
$sharedEvents->attach('Paste\ApiController', 'getLinks.post', function ($e) {
    $collection = $e->getParam('collection');

    // ... calculate $someOffset ...

    $links = $collection->getLinks();
    $prev  = new \PhlyRestfully\Link('prev');
    $prev->setRoute(
        'paste/api',
        array(),
        array('query' => array('offset' => $someOffset))
    );
    $links->add($prev);
});
```

This method could be extrapolated to add additional route parameters or options as well.

With these events, you have the ability to customize as needed. In most cases, however, if you can use paginators, do.

13.3 Query parameter white listing

Often when dealing with collections, you will use query string parameters to allow such actions as sorting, filtering, and grouping. However, by default, those query string parameters will not be used when generating links. This is by design, as the relational links in your resources typically should not change based on query string parameters.

However, if you want to retain them, you can.

As noted a number of times, the `ResourceController` exposes a number of events, and you can tie into those events in order to alter behavior. One method that the `HalCollection` class exposes is `setCollectionRouteOptions()`, which allows you to set, among other things, query string parameters to use during URL generation. As an example, consider this listener:

```
$allowedQueryParams = array('order', 'sort');
$sharedEvents->attach('Paste\ApiController', 'getList.post', function ($e) use ($allowedQueryParams)
    $request = $e->getTarget()->getRequest();
    $params  = array();
    foreach ($request->getQuery() as $key => $value) {
        if (in_array($key, $allowedQueryParams)) {
            $params[$key] = $value;
        }
    }
    if (empty($params)) {
        return;
    }

    $collection = $e->getParam('collection');
    $collection->setCollectionRouteOptions(array(
        'query' => $params,
    ));
});
```

The above is a very common pattern; so common, in fact, that we've automated it. You can whitelist query string parameters to use in URL generation for collections using the `collection_query_whitelist` configuration parameter for your resource controller:

```
return array(
    'phlyrestfully' => array(
        'resources' => array(
            'Paste\ApiController' => array(
                // ...
                'collection_query_whitelist' => array('order', 'sort'),
                // ...
            ),
        ),
    ),
);
```

CHAPTER

FOURTEEN

EMBEDDING RESOURCES

At times, you may want to embed resources inside other resources. As an example, consider a "user" resource: it may need to embed several addresses, multiple phone numbers, etc.:

HAL dictates the structure for the representation:

```
{
    "_links": {
        "self": {
            "href": "http://example.org/api/user/mwop"
        }
    },
    "id": "mwop",
    "full_name": "Matthew Weier O\'Phinney",
    "_embedded": {
        "url": {
            "_links": {
                "self": "http://example.org/api/user/mwop/url/mwop_net"
            },
            "url_id": "mwop_net",
            "url": "http://www.mwop.net/"
        },
        "phones": [
            {
                "_links": {
                    "self": "http://example.org/api/user/mwop/phones/1"
                },
                "phone_id": "mwop_1",
                "type": "mobile",
                "number": "800-555-1212"
            }
        ]
    }
}
```

However, what if our objects look like this:

```
class User
{
    public $id;
    public $full_name;

    /**
     * @var Url
```

```
     \*/
    public $url;

    /**
     * @var Phone[]
     \*/
    public $phones;
}

class Url
{
    public $url_id;
    public $url;
}

class Phone
{
    public $phone_id;
    public $type;
    public $number;
}
```

How, exactly, do we ensure that the `$url` and `$phones` properties are rendered as embedded resources?

The explicit way to handle it is within your listeners: assign the value of these properties to a `HalResource` or `HalCollection` (depending on whether they are single resources or collections of resources, respectively).

```
$user = $persistence->fetch($id);
$user->addresses = new HalResource($user->url, $user->url->url_id);
$user->phones    = new HalCollection($user->phones, 'api/user/phone');
```

From here, you can use the techniques covered in the *advanced routing*, *advanced rendering*, and *hydrators* sections to ensure that the various relational links are rendered correctly, and that the resources are properly rendered.

This is fairly straight-forward, but ultimately inflexible and prone to error. Many times, the properties will not be public, and in many circumstances, the setters will require specific, typed objects. As such, making a change like this will not work.

You can work around it by creating either a proxy resource object, or converting the resource to an array. However, there's a better way: *metadata maps*.

CHAPTER

FIFTEEN

METADATA MAPPING

If you have been reading the reference guide sequentially, almost every page has referred to this one at some point. The reason is that, for purposes of flexibility, PhlyRestfully has needed to provide a low-level, configurable mechanism that solves the problems of:

- ensuring resources have the correct "self" relational link
- ensuring resources are extracted to a JSON representation correctly
- ensuring that embedded resources are rendered as embedded HAL resources

To achieve this in a simpler fashion, PhlyRestfully provides the ability to create a "metadata map." The metadata map maps a class to a set of "rules" that define whether the class represents a resource or collection, the information necessary to generate a "self" relational link, and a hydrator to use to extract the resource.

This metadata map is defined via configuration. Let's consider the example from the *embedded resources section*:

```
return array(
    'phlyrestfully' => array(
        'metadata_map' => array(
            'User' => array(
                'hydrator'        => 'ObjectProperty',
                'identifier_name' => 'id',
                'route'           => 'api/user',
            ),
            'Url' => array(
                'hydrator'        => 'ObjectProperty',
                'route'           => 'api/user/url',
                'identifier_name' => 'url_id',
            ),
            'Phones' => array(
                'is_collection'   => true,
                'route'           => 'api/user/phone',
            ),
            'Phone' => array(
                'hydrator'        => 'ObjectProperty',
                'route'           => 'api/user/phone',
                'identifier_name' => 'phone_id',
            ),
        ),
    ),
);
```

Essentially, the map allows you to associate metadata about how the representation of a resource.

15.1 Metadata options

The following options are available for metadata maps:

- **hydrator**: the fully qualified class name of a hydrator, or a service name `Zend\Stdlib\Hydrator\HydratorPluginManager` recognizes, to use to extract the resource. (**OPTIONAL**)
- **identifier_name**: the resource parameter corresponding to the identifier; defaults to "id". (**OPTIONAL**)
- **is_collection**: boolean flag indicating whether or not the resource is a collection; defaults to "false". (**OPTIONAL**)
- **resource_route**: the name of the route to use for resources embedded as part of a collection. If not set, the route for the resource is used. (**OPTIONAL**)
- **route**: the name of the route to use for generating the "self" relational link. (**OPTIONAL**; this or `url` **MUST** be set, however)
- **route_options**: any options to pass to the route when generating the "self" relational link. (**OPTIONAL**)
- **route_params**: any route match parameters to pass to the route when generating the "self" relational link. (**OPTIONAL**)
- **url**: the specific URL to use with this resource. (**OPTIONAL**; this or `route` **MUST** be set, however)

15.2 Collections

If you paid careful attention to the example, you'll note that there is one additional type in the definition, `Phones`. When creating metadata for a collection, you need to define a first-class type so that `HalLinks` can match the collection against the metadata map. This is generally regarded as a best practice when doing domain modeling; a type per collection makes it easy to understand what types of objects the collection contains, and allows for domain-specific logic surrounding the collection.

However, that poses some problems if you want to *paginate your collection*, as instances of `Zend\Paginator\Paginator` are identified by `HalLinks` when rendering collections in order to create appropriate relational links.

The solution to that is to create an empty extension of `Paginator`:

```
1 use Zend\Paginator\Paginator;
2
3 class Phones extends Paginator
4 {
5 }
```

CHAPTER

SIXTEEN

THE API-PROBLEM LISTENER

In the chapter on *error reporting*, I noted that PhlyRestfully has standardized on the API-Problem format for reporting errors.

Currently, an API-Problem response will be created automatically for any of the following conditions:

- raising a `PhlyRestfully\Exception\CreationException` inside a `create` listener.
- raising a `PhlyRestfully\Exception\PatchException` inside a `patch` listener.
- raising a `PhlyRestfully\Exception\UpdateException` inside an `update` listener.
- raising an exception in any other `PhlyRestfully\Resource` event listener.

If the exception you raise implements `PhlyRestfully\Exception\ProblemExceptionInterface` – which `PhlyRestfully\Exception\DomainException` does, as does its descendents, the `CreationException`, `PatchException`, and `UpdateException` – you can set additional details in the exception instance before throwing it, allowing you to hit to `PhlyRestfully\ApiProblem` how to render itself.

There's another way to create and return an API-Problem payload, however: create and return an instance of `PhlyRestfully\ApiProblem` from any of the `Resource` event listeners. This gives you fine-grained control over creation of the problem object, and avoids the overhead of exceptions.

However, there's another way to receive an API-Problem result: raising an exception. For this the listener becomes important.

16.1 The Listener

`PhlyRestfully\Module` registers a listener with the identifier `PhlyRestfully\ResourceController` on its `dispatch` event. This event then registers the `PhlyRestfully\ApiProblemListener` on the application `render` event. Essentially, this ensures that the listener is only registered if a controller intended as a RESTful resource endpoint is triggered.

The listener checks to see if the `MvcEvent` instance is marked as containing an error. If so, it checks to see if the `Accept` header is looking for a JSON response, and, finally, if so, it marshals an `ApiProblem` instance from the exception, setting it as the result.

This latter bit, the `Accept` header matching, is configurable. If you want to allow an API-Problem response for other than the default set of mediatypes (`application/hal+json`, `application/api-problem+json`, and `application/json`), you can do so via your configuration. Set the value in the `accept_filter` subkey of the `phlyrestfully` configuration; the value should be a comma-separated set of mimetypes.

```php
return array(
    'phlyrestfully' => array(
        // ...
        'accept_filter' => 'application/json,text/json',
    ),
);
```

CHAPTER

SEVENTEEN

ALTERNATE RESOURCE RETURN VALUES

Typically, you should return first-class objects or arrays from your `Resource` listeners, and use the *hydrators map* and *metadata map* for hinting to PhlyRestfully how to render the various resources.

However, if you need to heavily optimize for performance, or want to customize your resource or collection instances without needing to wire more event listeners, you have another option: return `HalResource`, `HalCollection`, or `ApiProblem` instances directly from your `Resource` listeners, or the objects they delegate to.

17.1 HalResource and HalCollection

`PhlyRestfully\HalResource` and `PhlyRestfully\HalCollection` are simply wrappers for the resources and collections you create, and provide the ability to aggregate referential links. Links are aggregated in a `PhlyRestfully\LinkCollection` as individual `PhlyRestfully\Link` objects.

`HalResource` requires that you pass a resource and its identifier in the constructor, and then allows you to aggregate links:

```
use PhlyRestfully\HalResource;
use PhlyRestfully\Link;

// Create the HAL resource
// Assume $user is an object representing a user we want to
// render; we could have also used an associative array.
$halResource = new HalResource($user, $user->getId());

// Create some links
$selfLink = new Link('self');
$selfLink->setRoute('user', array('user_id' => $user->getId()));

$docsLink = new Link('describedBy');
$docsLink->setRoute('api/help', array('resource' => 'user'));

$links = $halResource->getLinks();
$links->add($selfLink)
      ->add($docsLink);
```

The above example creates a `HalResource` instance based on something we plucked from our persistence layer. We then add a couple of links describing "self" and "describedBy" relations, pointing them to specific routes and using specific criteria.

We can do the same for collections. With a collection, we need to specify the object or array representing the collection, and then provide metadata for various properties, such as:

- The route to use for generating links for the collection, including any extra routing parameters or options.
- The route to use for generating links for the resources in the collection, including any extra routing parameters or options.
- The name of the identifier key within the embedded resources.
- Additional attributes/properties to render as part of the collection. These would be first-class properties, and not embedded resources.
- The name of the embedded collection (which defaults to "items").

The following example demonstrates each of these options, as well as the addition of several relational links.

```
use PhlyRestfully\HalCollection;
use PhlyRestfully\Link;

// Assume $users is an iterable set of users for seeding the collection.
$collection = new HalCollection($users);

$collection->setCollectionRoute('api/user');
// Assume that we need to specify a version within the URL:
$collection->setCollectionRouteParams(array(
    'version' => 2,
));
// Tell the router to allow query parameters when generating the URI:
$collection->setCollectionRouteOptions(array(
    'query' => true,
));

// Set the resource route, params, and options
$collection->setResourceRoute('api/user');
$collection->setResourceRouteParams(array(
    'version' => 2,
));
$collection->setResourceRouteOptions(array(
    'query' => null, // disable query string params
));

// Set the collection name:
$collection->setCollectionName('users');

// Set some attributes: current page, total number of pages, total items:
$collection->setAttributes(array(
    'page'        => $page, // assume we have this from somewhere else
    'pages_count' => count($users),
    'users_count' => $users->countAllItems(),
));

// Add some links
$selfLink = new Link('self');
$selfLink->setRoute('api/user', array(), array('query' => true));
$docsLink = new Link('describedBy');
$docsLink->setRoute('api/help', array('resource' => 'users'));

$links = $collection->getLinks();
```

```
$links->add($selfLink)
    ->add($docsLink);
```

Using this approach, you can fully customize the `HalResource` and `HalCollection` objects, allowing you to set custom links, customize many aspects of output, and more. You could even extend these classes to provide additional behavior, and provide your own `HalLinks` implementation that renders them differently if desired.

The downside, however, is that it ties your implementation directly to the PhlyRestfully implementation, which may limit some use cases.

17.2 ApiProblem

Just as you can return a `HalResource` or `HalCollection`, you can also directly return a `PhlyRestfully\ApiProblem` instance if desired, allowing you to fully craft the return value.

Unlike `HalResource` and `HalCollection`, however, `ApiProblem` does not allow you to set most properties after instantiation, which means you'll need to ensure you have all your details up front.

The signature of the constructor is:

```
public function __construct(
    $httpStatus,                // HTTP status code used for the response
    $detail,                    // Summary of what happened
    $describedBy = null,        // URI to a description of the problem
    $title = null,              // Generic title for the problem
    array $additional = array() // Additional properties to include in the payload
);
```

Essentially, you simply instantiate and return an `ApiProblem` from your listener, and it will be used directly.

```
use PhlyRestfully\ApiProblem;

return new ApiProblem(
    418,
    'Exceeded rate limit',
    $urlHelper('api/help', array('resource', 'error_418')),
    "I'm a teapot",
    array(
        'user'  => $user,
        'limit' => '60/hour',
    )
);
```

And with that, you have a fully customized error response.

CHAPTER

EIGHTEEN

CHILD RESOURCES

Resources often do not exist in isolation. Besides some resources embedding others, in some cases, a resource exists only as a result of another resource existing – in other words, within a hierarchical or tree structure. Such resources are often given the name "child resources."

In the *advanced routing chapter*, we looked at one such example, with a user and addresses.

```
'users' => array(
    'type' => 'Segment',
    'options' => array(
        'route' => '/users[/:user_id]',
        'controller' => 'UserResourceController',
    ),
    'may_terminate' => true,
    'child_routes' => array(
        'addresses' => array(
            'type' => 'Segment',
            'options' => array(
                'route' => '/addresses[/:address_id]',
                'controller' => 'UserAddressResourceController',
            ),
        ),
    ),
),
```

In that chapter, I looked at how to tie into various events in order to alter routing parameters, which would ensure that the relational URLs were generated correctly. I also noted that there's a better approach: *metadata maps*. Let's look at such a solution now.

First, let's make some assumptions:

- Users are of type `User`, and can be hydrated using the `ClassMethods` hydrator.
- Individual addresses are of type `UserAddress`, and can by hydrated using the `ObjectProperty` hydrator.
- Address collections have their own type, `UserAddresses`.
- The users collection is called "users"
- The address collection is called "addresses"
- The class `UserListener` listens on `Resource` events for users.
- The class `UserAddressListener` listens on `Resource` events for addresses.

Now, let's create some resource controllers, using configuration as noted in the chapter on *resource controllers*.

```
return array(
    // ...
    'phlyrestfully' => array(
        'resources' => array(
            'UserResourceController' => array(
                'listener'                => 'UserListener',
                'collection_name'         => 'users',
                'collection_http_options' => array('get', 'post'),
                'resource_http_options'   => array('get', 'patch', 'put', 'delete'),
                'page_size'               => 30,
            ),
            'UserAddressResourceController' => array(
                'listener'                => 'UserAddressListener',
                'collection_name'         => 'addresses',
                'collection_http_options' => array('get', 'post'),
                'resource_http_options'   => array('get', 'patch', 'put', 'delete'),
            ),
        ),
    ),
);
```

Now we have controllers that can respond properly. Let's now configure the metadata and hydrator maps for our resources.

```
return array(
    // ...
    'phlyrestfully' => array(
        // ...
        'metadata_map' => array(
            'User' => array(
                'hydrator'        => 'ClassMethods',
                'identifier_name' => 'user_id',
                'route'           => 'users',
            ),
            'UserAddress' => array(
                'hydrator'        => 'ObjectProperty',
                'identifier_name' => 'address_id',
                'route'           => 'users/addresses',
            ),
            'UserAddresses' => array(
                'identifier_name' => 'address_id',
                'route'           => 'users/addresses',
                'is_collection'   => true,
                'route_options'   => array('query' => true),
            ),
        ),
    ),
);
```

Now, when we render a `User`, if it composes a `UserAddresses` object, that object will be rendered as an embedded collection, and each resource inside it will be rendered using the appropriate route and identifier.

CHAPTER

NINETEEN

CLASSES AVAILABLE

This is a partial list of classes only; for a full list of classes, visit the API documentation.

Classes below are linked to their API documentation. The classes listed are chosen because they are either directly discussed in this documentation, or because they may be of interest as extension points.

- ApiProblem
- HalCollection
- HalResource
- Link
- LinkCollection
- Metadata
- MetadataMap
- Resource
- ResourceController
- ResourceEvent
- HalLinks

CHAPTER

TWENTY

RESTFUL JSON API PRIMER

CHAPTER

TWENTYONE

PHLYRESTFULLY WALKTHROUGH

CHAPTER

TWENTYTWO

REFERENCE GUIDE

CHAPTER

TWENTYTHREE

API DOCS:

API docs are available.

CHAPTER

TWENTYFOUR

INDICES AND TABLES

- *Welcome to PhlyRestfully!*
- *genindex*
- *search*