RESTful Services made easy with ZF2

by Rob Allen and Matthew Weier O'Phinney

January 2013

About us

Rob Allen

- ZF community team
- @akrabat

Matthew Weier O'Phinney

- ZF project lead
- @mwop

Agenda

- RESTful fundamentals
- Zend Framework 2 fundamentals
- RESTful ZF2

Restful fundamentals

REpresentational State Transfer

What is Rest?

- An *architecture*
- Centers on the transfer of *representations* of *resources*
    - A *resource* is any concept that can be addressed
    - A *representation* is typically a document that captures the current or intended state of a resource
- A *client* makes requests of a server when it wants to transition to a new state

Strengths

- Loose coupling
- Less typing (counter-example: SOAP)
- Emphasis on readability; uses nouns and verbs
    - HTTP methods as verbs: GET, POST, PUT, DELETE, etc.
    - Resources as nouns, and, further, collections

Constraints

- Client/Server
    - Clients are not concerned with storage, allowing them to be portable.
    - Servers are not concerned with UI or user state, allowing scalability.

Constraints

- Stateless
    - No client context stored between requests. This means no sessions!

Constraints

- Cacheable
  - Non-idempotent methods should allow clients to cache responses.
  - Clients should honor HTTP headers with respect to caching.

Constraints

- Layered system
    - Client should not care whether it is connected directly to the server, or to an intermediary proxy.

Constraints

- Uniform Interface
    - Identification of resources
    - Manipulation of resources through representations
    - Self-descriptive messages
    - Hypermedia as the engine of application state (HATEOAS)

Primary aspects of a RESTful web service

- Base URI for *each* resource: http://status.dev:8080/api/status/matthew
- Media type used for representations of the resource
- HTTP methods are the set of operations allowed for the resource
- The API must be hypertext driven (i.e., provide links for allowed state transitions)

Content negotiation

- Correctly parse the request
    - Read the `Content-Type` header
    - Raise “415 Unsupported media type” status if unsupported
- Correctly create the response
    - Read the `Accept` header
    - Set the `Content-Type` header

Hypermedia

- What is it?
- Why is it important?

What is hypermedia?

- Media type used for a representation
- The link relations between representations and/or states

Why is hypermedia important?

- Discoverability

JSON and Hypermedia

JSON does not have a defined way of providing hypermedia links

Options:

- “Link” header (GitHub approach)
- application/collection+json
- application/hal+json

Link header

```
Link: <url>; rel="relation"[, ... ]
```

application/collection+json

See http://amundsen.com/media-types/collection/format/

```
{
    "collection":
    {
        "links": [
            {"href": "<uri>", "rel": "relation"}
        ]
    }
}
```

application/hal+json

See http://tools.ietf.org/html/draft-kelly-json-hal-03

```
{
    "_links": {
        "relation": {"href": "<uri>"}
    }
}
```

Error reporting

- HTTP status: 4xx, 5xx
- No further information!
- Solution
    - application/api-problem+json

application/api-problem+json

See
http://tools.ietf.org/html/draft-nottingham-http-problem-02

```
{
    "describedBy": "<url>",
    "title": "generic title of error type",
    "httpStatus": <status code>,
    "detail": "specific message detailing error"
}
```

Documenting your API

- What operations are available for a given resource?
- What do representations look like? How do I need to form my request? What modifiers might be available?

OPTIONS

- Minimally, respond to `OPTIONS` requests, indicating HTTP methods allowed via the `Allow` header.
- Potentially include information in the HTTP body.
    - http://zacstewart.com/2012/04/14/http-options-meth
    - http://vimeo.com/49613738 ("Fun with OPTIONS" by D. Keith Casey at REST Fest 2012)

Using a “describedby” Link relation

- Use a `Link` header with a “describedby” link relation pointing to documentation. See http://www.mnot.net/blog/2012/10/29/NO_OPTIONS

```
Link: <http://status.dev/api/status/docs.md>; \
 rel="describedby"
```

ZF2 Fundamentals

The next generation of Zend Framework

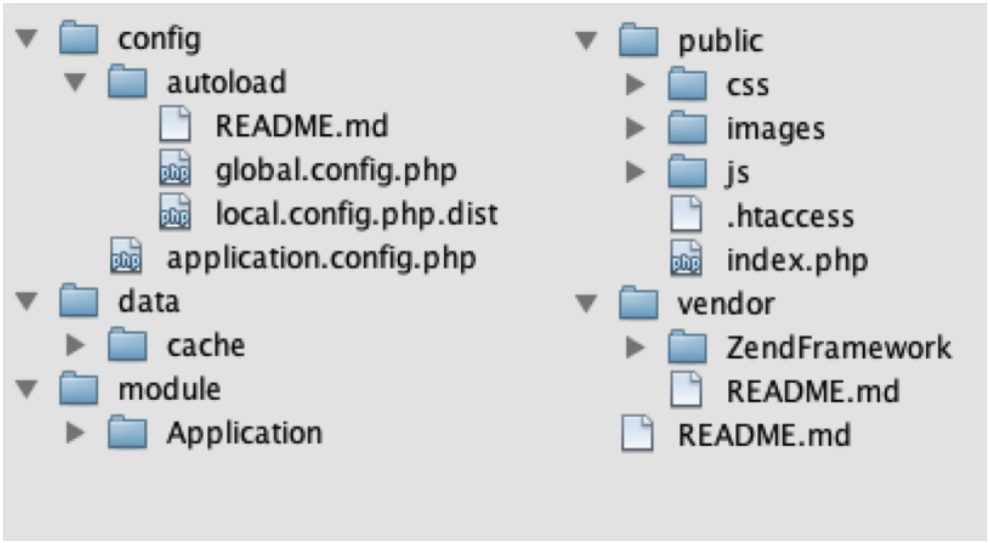
Highlights

- PHP 5.3.3+ (and tested on 5.4, as well as upcoming 5.5)
- Modular & flexible (ModuleManager)
- Event-driven (EventManager)
- Leverage Inversion of Control (ServiceManager)
- Re-written MVC, Forms, I18n, Db, and more

MVC key features

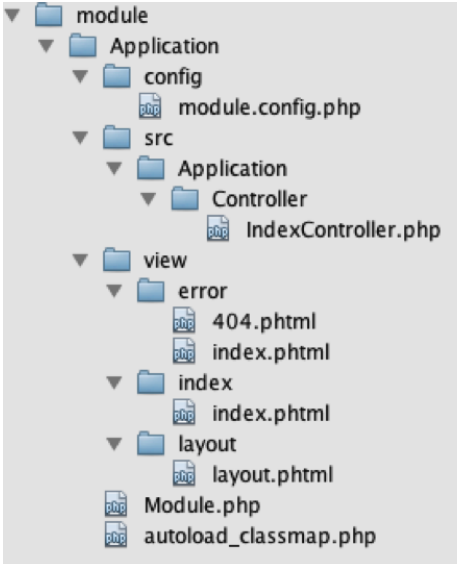
- Everything is in a Module
- MVC is event driven and uses ServiceManager
- Controllers contain actions
    - which return data for your view scripts, or a response
- View scripts contain display code (e.g. HTML)

Directory structure



```
▼ config
    ▼ autoload
          README.md
          global.config.php
          local.config.php.dist
      application.config.php
▼ data
    ▶ cache
▼ module
    ▶ Application
▼ public
    ▶ css
    ▶ images
    ▶ js
      .htaccess
      index.php
▼ vendor
    ▶ ZendFramework
      README.md
  README.md
```

Module directory



```
module
  Application
    config
      module.config.php
    src
      Application
        Controller
          IndexController.php
    view
      error
        404.phtml
        index.phtml
      index
        index.phtml
      layout
        layout.phtml
    Module.php
    autoload_classmap.php
```

Events

Publish and listen to events

Events

- An object *triggers* an *event*
- Other objects *listen* to *events*

Terminology

- An **EventManager** is an object that holds a collection of listeners for one or more named events, and which triggers events.
- An **event** is an action.
- A **listener** is a callback that can react to an event.
- A **Target** is an object that creates events

Simple example

```
use Zend\EventManager\EventManager,
    Zend\EventManager\Event;

$callback = function($event) {
    echo "An event has happened!\n";
    var_dump($event->getName());
};
$events = new EventManager();
$events->attach('eventName', $callback);

echo "\nRaise an event\n";
$events->trigger('eventName');
```

Listeners

Just a function (Any callback)

```
$callback = function($event) {
    echo "An event has happened!\n";
    var_dump($event->getName());
    var_dump($event->getTarget());
    var_dump($event->getParams());
};

$events = $someObject->getEventManager();
$events->attach('eventName', $callback);
```

The target

Compose an EventManager within a class…

```
use Zend\EventManager\EventManager,
    Zend\EventManager\Event;

class MyTarget
{
  public $events;

  public function __construct()
  {
    $this->events = new EventManager();
  }
  //...
```

The target

… and trigger actions within methods.

```php
public function doIt()
{
  $event = new Event();
  $event->setTarget($this);
  $event->setParam('one', 1);
  $this->events->trigger('doIt.pre', $event);

  // do something here

  $this->events->trigger('doIt.post', $event);
}
```

Typical usage

```
$callback = function ($event) {
    echo "Responding to doIt.pre!\n";
    var_dump(get_class($event->getTarget()));
    var_dump($event->getName());
    var_dump($event->getParams());
};


$target = new MyTarget();
$target->events->attach('doIt.pre', $callback);
$target->doIt();
```

Attaching listeners globally

- Listeners are used for cross-cutting concerns
- You want to set up listeners before you instantiation of object with event manager
- For example: logging, caching

SharedEventManager

Attach a listener to another class' event manager

```
$shared = $events->getSharedManager();
// or
$shared = StaticEventManager::getInstance();

$shared->attach('Gallery\\Mapper\\Photo',
    'findById.pre', function(Event $e) {
        $id = $e->getParam('id');
        $message = "Retrieving photo: $id";
        MyLogger::log($message);
});
```

Returned values from listeners

```
public function doIt()
{
  $events = $this->events;
  $results = $events->trigger('doIt', $this);
  foreach ($results as $result) {
    var_dump($result);
  }
}
```

`$results` are in *reverse order*
(most recently triggered event first)

Short-circuiting

```
$params = array('id' => 1);
$results = $this->events->trigger('doIt.pre',
    $this, $params, function ($result) {
      if ($result instanceof ResultSet) {
        return true;
      }
      return false;
    }
  );

if ($results->stopped()) { // We ended early }
```

Priority

- Control the order of execution of listeners
- *$priority* is last parameter to *attach()*

```
$events->attach('doIt.pre', $cb, $priority);
```

- Default is 1
    - Larger number increases priority (e.g. 1000)
    - Smaller number decreases priority (e.g. -500)

Services

It (lazily) instantiates and holds objects.

Services

- Objects you work with (including Controllers).
- Easy to replace alternative implementations.
- Clean and simple way to configure dependencies.
- Explicit and easy to understand - no magic!
- Inversion of Control.

Usage

```
$controller = $sm->get('Gallery\Mapper\Photo');
```

Types of services

- Instances (`services`)
- Constructor-less classes (`invokables`)
- Factories for objects with dependencies (`factories`)
- Aliased services (`aliases`)
- Automated initialization (`initializers`)
- Factories for multiple related objects (`abstract_factories`)

Instances

```
// programatically
$sm->setService('foo', $fooInstance);

// configuration
array('services' => array(
    'foo' => new Foo(),
));
```

Invokables

```
// programatically
$sm->setInvokableClass('foo', 'Bar\Foo');

// configuration
array('invokables' => array(
    'foo' => 'Bar\Foo',
));
```

Factories

```
// programatically
$sm->setFactory('foo', function($sm) {
        $dependency = $sm->get('Dependency')
        return new Foo($dependency);
    });

// configuration
array('factories' => array(
  'foo' => function($sm) { //.. },
  'bar' => 'Some\Static::method',
  'baz' => 'Class\Implementing\FactoryInterface',
  'bat' => 'Class\Implementing\Invoke',
));
```

Aliases

```
// programatically
$sm->setAlias('foo_db', 'db_adapter');

// configuration
array('factories' => array(
    'foo_db', 'db_adapter', // alias of a service
    'bar_db', 'foo_db',     // alias of an alias
));

// All the same instance
$db = $sm->get('db_adapter');
$db = $sm->get('foo_db');
$db = $sm->get('bar_db');
```

Initializers

```
// programatically
$sm->addInitializer($callback);

// configuration
array('initializers' => array(
  $instance,
  $callback,
  'Class\Implementing\InitializerInterface',
  'Class\Implementing\Invoke',
));
```

An initializer

```
function($instance, $sm) {
    if ($instance instanceof FooAwareInterface) {
        return;
    }
    $instance->setFoo($sm->get('foo'));
},
```

Abstract factories

Factory capable of handling multiple services

```
// programatically
$sm->addAbstractFactory($abstractFactoryInstance);
$sm->addAbstractFactory('FooFactory');

// configuration
array('abstract_factories' => array(
  'Class\Implementing\AbstractFactoryInterface',
    $someAbstractFactoryInstance,
);
```

An abstract factory

```
class AFactory implements AbstractFactoryInterface
{
    public function canCreateServiceWithName(
        ServiceLocatorInterface $services,
        $name, $requestedName
    ) {
        return in_array($name, array('foo','bar'));
    }
    public function createServiceWithName(/*sig*/)
    {
        return $name == 'foo' ? new Foo : new Bar;
    }
}
```

Other features

- All plugin managers are services managers.
- Services are shared - can disable per service.
- Manager “peering” is available.

Configuration in practice

- A nested array in:
    - MyModuleModule::getServiceConfig()
    - ‘service_manager’ array key in config
- sub-array keys : `services`, `invokables`, `factories`, `aliases`, `initializers`, `abstract_factories`

Modules

Re-usable pieces of functionality for constructing a more complex application.

Modules

Provide your application with:

- autoloading
- configuration
- services (inc controllers, plugins, etc.)
- event listeners

Reusable between applications - “plug & play”!

What can modules be?

Anything!

- *Plugins:* payment module for e-commerce
- *View helpers:* Markdown support
- *Themes:* CSS files, images, view scripts
- *Libraries:* Doctrine2 integration, RESTful support
- *Applications:* blog, e-commerce platform, CMS

A module is…

- A PHP namespace
- A class called `Module` within that namespace
    - which provides features to the application

A ZF2 Module

```php
<?php
namespace MyModule;

class Module {}
```

That’s it.

A *complete* ZF2 module

```
namespace EdpMarkdown;
class Module extends
    \Zend\View\Helper\AbstractHelper
{
    public function getViewHelperConfig() {
        return array('services' => array(
            'markdown' => $this));
    }
    public function __invoke($string = null) {
        require_once __DIR__ . 'markdown.php';
        return Markdown($string);
    }
}
```

ModuleManager

- Loads all modules
- Triggers an event for each module
    - allowing **listeners** to act on Module classes
    - Results in calls specific methods within your `Module` class

Module methods called

- `getAutoloaderConfig()`
- `init()`
- `onBootstrap()`
- Service Manager methods:
    - `getServiceConfig()`
    - `getControllerConfig()`
    - `getControllerPluginConfig()`
    - `getViewHelperConfig()`

Other actions

- If `LocatorRegisteredInterface` is implemented, then register with the service manager.
- All configs are merged together:

  1. `getConfig()` results merged in the order modules are loaded.
  2. Config glob/static paths are merged.
  3. The `getServiceConfig()` (and friends) results are merged together then merged with the result of steps 1 and 2.

A typical Module class

```
namespace My;
class Module {
    public function getAutoloaderConfig() {
        // return config for autoloader factory
    }
    public function getConfig() {
        return include
            __DIR__ . '/config/module.config.php';
    }
    public function onBootstrap($e) {
        // do initialization
    }
}
```

Module best practices

- Keep `init()` and `onBootstrap()` very lightweight.
- Read-only (*do not perform writes within modules*).
- Utilize a vendor prefix (e.g., `EdpMarkdown`, not `Markdown`).
- Do one thing, and do it well.

RESTful ZF2

Putting REST & ZF2 together

Foundations

- Routing
- `AbstractRestfulController`
- Reacting to request headers
- Creating hypermedia payloads
- Creating error payloads

Routing

- Route to an `AbstractRestfulController` implementation
    - Allows a single route to manage all HTTP methods for a given resource
- Use a combination of Literal and/or Segment routes

Sample Route

```
'status' => array(
    'type' => 'Segment',
    'options' => array(
        'route' => '/api/status[/:id]',
        'defaults' => array(
            'controller' => 'StatusController',
        ),
        'constraints' => array(
            'id' => '[a-f0-9]{40}',
        ),
    ),
),
```

AbstractRestfulController

- Maps HTTP methods to individual class methods
- Performs basic content-negotiation (`application/www-form-urlencoded` and JSON bodies will be parsed and provided as `$data`)

Mapping methods

- `GET` :: `getList()` or `get($id)`
- `POST` :: `create($data)`
- `PUT` :: `replaceList()`, `update($id, $data)`
- `PATCH` :: `patch($id, $data)`
- `DELETE` :: `deleteList()`, `delete($id)`
- `HEAD` :: `head($id = null)`
- `OPTIONS` :: `options()`

Selecting an acceptable view model

- Select a view model based on `Accept`
- Attach a view strategy based on view model

AcceptableViewModelSelector

- Controller plugin

```
$criteria = array(
    'Zend\View\Model\JsonModel' => array(
        '\*/json',
    ),
);
$model = $this->acceptableViewModel($criteria);
```

Changing view strategy based on model

- Listen on the controller's dispatch event

```
$sharedEvents->attach(
'Zend\Mvc\Controller\AbstractRestfulController',
'dispatch',
$listener
-10
);
```

Sample listener

```
function (MvcEvent $e) {
  $result = $e->getResult();
  if (!$result instanceof JsonModel) {
      return;
  }
  $app      = $e->getApplication();
  $services = $app->getServiceManager();
  $strategy = $services->get('ViewJsonStrategy');
  $view     = $services->get('View');
  $view->attach($strategy, 100);
},
```

Directly examining the Accept header

```
$headers = $request->getHeaders();
if (!$headers->has('Accept')) {
    // no Accept header; do default
    return;
}
$accept = $headers->get('Accept');
if ($accept->match($mediaType)) {
    // we have a match!
    return;
}
```

Hypermedia payloads

- Links should be fully qualified: include, scheme, server, and port if necessary
- A `self` relation is recommended
- With paginated sets, include `first`, `last`, `next`, and `prev` relations

Tools for creating links

- The `url` controller plugin and/or view helper can generate the path if a route is known.
- The `serverUrl` view helper can generate the scheme/server/port combination
- Paginators can be inspected and used to generate pagination relations

Generating individual links

```
$path = $urlHelper->fromRoute($route, array(
    'id' => $id,
));
$url  = $serverUrlHelper->__invoke($path);
```

Generating paginated links

```
// $page is the current page
// $count is the total number of pages
// $base is the base URL to the resource
$next  = ($page == $count) ? false : $page + 1;
$prev  = ($page == 1)      ? false : $page - 1;
$links = array(
    'self' => $base
        . (1 == $page ? '' : '?p=' . $page),
);
if ($page != 1) {
    $links['first'] = $base;
}
```

cont...

```
if ($count != 1) {
    $links['last'] = $base . '?p=' . $count;
}
if ($prev) {
    $links['prev'] = $base
        . ((1 == $prev) ? '' : '?p=' . $prev;
}
if ($next) {
    $links['next'] = $base . '?p=' . $next;
}
```

Where to generate links

- Controller is easiest, but may not be semantically correct
- View model makes sense, but is hard to inject with helpers
- Renderer makes sense, but likely requires specialized payloads in the view model
- A event listener could process the view model and inject them; similar issues to the renderer, though.
- Choose your poison.

Error payloads

- Be consistent
- Provide detail
- `application/api-problem+json` is a nice standard

API-Problem payloads

- `describedby` is required. If corresponding to HTTP status, http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html describing HTTP status codes is a nice default.
  - `title` is also required; again, if corresponding to HTTP status, use established status descriptions.
  - `httpStatus` is not required, but recommended.
  - `detail` is your place to provide any additional information.

Where to generate API-Problem payloads

- Typically, within the controller; this is where the errors happen.
- You may also want listeners on `dispatch.error` so you can generate 404 responses in this format.

Practical application

- **YOU** will build a simple “status” API for posting social status
    - “text” representing the status
    - “user” representing the user posting the status
    - “timestamp” when the status was created
    - Collection of statuses by user, in reverse chronological order
        - User is present in the URI

Steps

- Create the domain logic (this is the hard part)
- Create a route
- Create a controller that:
    - calls on the domain logic
    - varies the view model based on the `Accept` header
    - creates API-Problem payloads for errors
- Create a listener for injecting hypermedia links in the view model

Route

- `/status/:user[/:id]`

Controller

- Extend `AbstractRestfulController`
    - use `AcceptableViewModelSelector` to pull a relevant view model based on `Accept` header; create a special view model type that we can listen for later.
    - set specific variables in the view that we can query later
    - use a special object for indicating errors
    - set appropriate HTTP status codes

Listener

- Listen for our special view model type
- If an error is detected:
    - Create an API-Problem payload
    - Set the response status code
- Generate hyperlinks based on whether we have a collection or an individual item.

Demonstration

This is meant to be alive demo of the finished API, and maybe some code samples.

PhlyRestfully

- Module that does these bits for you
- Add it to composer
    - "phly/phly-restfully": "dev-master@dev"
- Provide a resource listener that does the various persistence related operations and a route, and go.

Review

What have we learnt today?

Review

- REST is an architecture, with lots of recommendations but no single, canonical methodology
- Don’t skimp or skip the documentation!

Review

- REST has lots of little details to pay attention to:
    - URIs *per resource*
    - HTTP methods indicating the *operations* available for a resource
    - Media types indicating resource *representations* govern how to parse a request as well as how to format a response
    - Hypermedia links to promote discoverability and available state changes

Review

- Several emerging standards surrounding specifically RESTful **JSON** APIs
    - Collection+JSON
    - Hypertext Application Language (HAL)
    - API-Problem

Review

- ZF2 has a lot of built-in features to help build RESTful applications
    - `AbstractRestfulController`
    - `Accept` header implementation
    - Rich HTTP tooling in general
    - Flexible view layer

Thank you!

https://joind.in/7781

Rob Allen : http://akrabat.com : @akrabat
Matthew Weier O'Phinney : http://mwop.net : @mwop