Installation

To start serving RESTful HTTP requests, you need to go through three simple steps:

1. Configure virtualhost in your web-server so that all HTTP requests to "non-existant" "files" are sent to your PHP file, say: api.php (see: Appendix A)
2. Create api.php where you instantiate and configure a router object
3. Write controller callbacks.

A Simple Router

For a very simple case of getting specific user object, the code of api.php would look something like:

```
<?php

require_once(__DIR__ . '/zaphpa/zaphpa.lib.php');
$router = new Zaphpa_Router();

$router->addRoute(array(
  'path'     => '/users/{id}',
  'handlers' => array(
    'id'          => Zaphpa_Constants::PATTERN_DIGIT, //enforced to be numeric
  ),
  'get'      => array('MyController', 'getPage'),
));

try {
  $router->route();
} catch (Zaphpa_InvalidPathException $ex) {
  header("Content-Type: application/json;", TRUE, 404);
  $out = array("error" => "not found");
  die(json_encode($out));
}
```

In this example, {id} is a URI parameter of the type "digit", so `MyController->getPage()` function will get control to serve URLs like:

- http://example.com/users/32424
- http://example.com/users/23

However, we asked the library to ascertain that the {id} parameter is a number by attaching a validating handler: "Zaphpa*Constants::PATTERN*DIGIT" to it. As such, following URLs will not be handed over to the `MyController->getPage()` callback:

- http://example.com/users/ertla
- http://example.com/users/asda32424
- http://example.com/users/32424sfsd
- http://example.com/users/324sdf24

Simple Callbacks

A callback can be a simple PHP function. In most cases it will probably be a method on a class. Callbacks are passed two arguments:

1. `$req` is an object created and populated by Zaphpa from current HTTP request.
2. `$res` is a response object. It is used by your callback code to incrementally assemble a response, including both the response headers, as well as: the response body.

We will look into the details of $req and $res objects further in the documentation. Following are some example callback implementations:

```
<?php

class MyController {

  public function getPage($req, $res) {
    $res->setFormat("json");
    $res->add(json_encode($req->params));
    $res->add(json_encode($req->data));
    $res->send(301);
  }

  public function postPage($req, $res) {
    $res->add(json_encode($req->params));
    $res->add(json_encode($req->data));
    $res->send(201, 'json');
  }

}
```

Request Object

`$req (request)` object contains data parsed from the request, and can include properties like:

1. `$params` - which contains all the placeholders matched in the URL (e.g. the value of the "id" argument)
2. `$data` - an array that contains HTTP data. In case of HTTP GET it is: parsed request parameters, for HTTP POST, PUT and DELETE requests: data variables contained in the HTTP Body of the request.
3. `$version` - version of the API if one is versioned (not yet implemented)
4. `$format` - data format that was requested (e.g. XML, JSON etc.)

Following is an example request object:

```
<?php
```

```
Zaphpa_Request Object
(
  [params] => Array
    (
      [id] => 234234
    )
  [data] => Array
    (
      [api] => 46546456456
    )
  [formats] => Array
    (
      [0] => text/html
      [1] => application/xhtml+xml
      [2] => application/xml
    )
  [encodings] => Array
    (
      [0] => gzip
      [1] => deflate
      [2] => sdch
    )
  [charsets] => Array
    (
      [0] => ISO-8859-1
      [1] => utf-8
    )
  [languages] => Array
    (
      [0] => en-US
      [1] => en
    )
  [version] =>
  [method] => GET
  [clientIP] => 172.30.25.142
  [userAgent] => Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/535.2...
  [protocol] => HTTP/1.1
)
```

Request parsing

Zaphpa provides multiple shortcuts to make request-parsing easier and less error-prone. One such shortcut is `$req->get_var(<varname>)` function.

1. `$req->get_var('varname')` - since $req object populates `$data` object, you can access request variables (query variables for GET or HTTP Body data for PUT/POST/DELETE) through the $data array directly. However am HTTP client may not set a variable that your callback expects, causing PHP to throw a warning. Instead of having your callback code check each call to `$req->data['varname']` on being empty Zaphpa provides a convenience method: $req->get*var('varanme'). get*var() returns value of the HTTP variable if it is set, or null, otherwise.

Response Object

`$res (response)` object is used to incrementally create content. You can add chunks of text to the output buffer by calling: `$res->add (String)` and once you are done you can send entire buffer to the HTTP client by issuing: `$res->send(<HTTP_RESPONSE_CODE>)`. HTTP*RESPONSE*CODE is an optional parameter which defaults to (you guessed it:) 200.

Response object is basically an output buffer that you can keep adding output chunks to, while you are building a response. Following methods are available on the response class:

1. `$res->add($string)` - adds a string to output buffer
2. `$res->flush($code, $format)` - sends current output buffer to client. Can take optional output code (defaults to 200) and output format (defaults to request format) arguments. **Caution**: obviously you can/should only indicate $code or $format, the first time you invoke the method, since these values can not be set once output is sent to the client.
3. `$res->send($code, $format)` - sends current output buffer to the client and terminates response.

Static Accessors for Req/Res

The request and response objects for the current HTTP process are passed to the callback function as arguments, but sometimes you need to access them from code much deeper in your codebase (e.g. models). To make this easy and avoid the necessity of extraneous passing around of $req and $res, Zaphpa provides two convenience static functions:

```
Zaphpa_Router::request() - returns $req
Zaphpa_Router::response() - returns $res
```

Middleware

One of the powerful features of Zaphpa is the ability to intercept request and perform centralized pre- and post- processing of a request. To hook into request processing flow, register your middleware implementation with the router:

```
$router->attach('MyMiddlewareImpl');
```

Where `MyMiddlewareImpl` is the class name of an implementation of a Zaphpa_Middleware abstract class. You can implement following methods, in your middleware:

- `->preprocess(&$route)` - hooks very early into the process and allows adding custom route mappings.
- `->preroute(&$req, &$res)` - hooks into the process once request has been analyzed, route handler has been identified, but before route handler
- `->prerender(&$buffer)` - gets a chance to alter buffer right before it is assembled for output. Please note that this method may be called multiple times, to be more precise: every time your callback function calls `->flush` and tries to output a chunk of response buffer to HTTP, `->prerender` will get a chance to alter the buffer.

Middleware is an excellent place to implement central authentication, authorization, versioning and other infrastructural features.

An example implementation (however meaningless) of a middleware can be found in Zaphpa tests:

```
<?php
```

```
class ZaphpaTestMiddleware extends Zaphpa_Middleware {
  function preprocess(&$router) {
    $router->addRoute(array(
          \<em\>'path'     => '/middlewaretest/{mid}',\</em\>
          'get'      => array('TestController', 'getTestJsonResponse'),
    ));
  }

  function preroute(&$req, &$res) {
    // you get to customize behavior depending on the pattern being matched in the cu
rrent request
    if (self::$context['pattern'] == '/middlewaretest/{mid}') {
      $req->params['bogus'] = "foo";
    }
  }

  function prerender(&$buffer) {
      $dc = json_decode($buffer[0]);
      $dc->version = "2.0";
      $buffer[0] = json_encode($dc);
  }
}
```

Middleware Context

Please note the usage of `self::$context['pattern']` variable in the `->preroute` method. Often `preroute` needs to modify behavior based on the current URL Route being matched. The variable `self::$context['pattern']` carries that pattern. Please make sure to match it with the exact definition(s) in your routes configurations.

Full list of variables exposed through context:

- `pattern` - URI pattern being matched (in the format it was defined in the routes configuration, includes placeholders)
- `http_method` - current HTTP Method being processed.
- `callback` - callback in PHP format i.e.: name of the function or an array containing classname and method name.

Middleware Route Restrictions

As we saw in the example above, frequently you may want to only enable your middleware for certain routes. Instead of hard-coding that logic as part of the middleware implementation, Zaphpa makes it easy to declaratively set the scope of Middleware activity ("restrict" middleware execution to only certain routes):

```
<?php

$router->attach('MyMiddleWare')
       ->restrict('preroute', 'GET', '/users')
       ->restrict('prerender', array('POST', 'GET'), '/tags')
       ->restrict('preroute', '*', '/groups');
```

In the example above:

- first argument indicates which hook of the middleware you want to restrict routes for. Out of three currently supported hooks, you can restrict "preroute" and "prerender", but you can not restrict "preprocess", because preprocess hook is typically used to modify routes list and it does not make any sense to have route restrictions at that point of routing execution.
- the second argument indicates which HTTP methods you want to restrict (can be a single string, an array, or "*" which indicates: all http methods)
- third argument is a URL path you want to restrict middleware execution to. `->restrict()` invocations can be chained in a jQuery-like syntax, which you see above.

Prebuilt Middleware

HTTP Method Overrides

In REST you typically operate with following common HTTP Methods ("verbs" for CRUD): GET, PUT, POST, DELETE (and sometimes: PATCH). Using these methods can be problematic, in certain cases however. Some HTTP Proxies often block any methods but GET AND POST, as well as: making cross-domain Ajax calls with custom verbs can be hard.

Zaphpa implements common HTTP Method override trick to allow an effective solution. You can still implement proper HTTP Methods and if clients have problem making a particular HTTP Method-based request, they can make HTTP POST instead, and indicate the method they "meant" in request headers with the "X-HTTP-Method-Override" header.

Method override is a middleware plugin that is disabled by default. To enable it, add following line to your router initialization code:

```
$router->attach('MethodOverride');
```

Auto-Documentator

On to more useful middleware implementation, Zaphpa comes with an auto-documentation middleware plugin. This plugin creates an endpoint that can list all available endpoints, even parse PHPDoc comments off your code to provide additional documentation. To enable the middleware:

```
$router->attach('ZaphpaAutoDocumentator');
```

which will create documentation endpoint at: '/docs'. If you would rather create endpoint at another URI:

```
$router->attach('ZaphpaAutoDocumentator', '/apidocs');
```

If you want documentation to also show filename, class and callback method for each endpoint:

```
$router->attach('ZaphpaAutoDocumentator', '/apidocs', $details = true);
```

If you don't want some endpoints to be exposed in the documentation (say, for security reasons) you can easily hide those by adding `@hidden` attribute to the PHP docs of the callback for the endpoint. To build a more elaborate authorization schema, you would need to implement a custom middleware, but it's certainly possible.

Beware: problem has been reported when trying to use doc comment parsing in PHP with eAccelerator. There seems to be no such problem with APC.

Ajax-friendly Endpoints

As you probably know, Ajax calls can not normally access API URLs on another domain (or even another port of the same domain, actually). This is a problem sometimes solved using JSONP. We think a better solution

is: Cross-Origin Resource Sharing (CORS). Zaphpa comes with a simple pre-built middleware plugin to enable CORS for all endpoints. To enable CORS for any domain:

```
$router->attach('ZaphpaCORS');
```

or if you want to enable CORS only for specific domain(s):

```
$router->attach('ZaphpaCORS', 'http://example.com http://foo.example.com');
```

If you want to enable CORS only for specific routes:

```
<?php

$router->attach('ZaphpaCORS', '*')
       ->restrict('preroute', 'GET', '/users')
       ->restrict('preroute', array('POST', 'GET'), '/tags')
       ->restrict('preroute', '*', '/groups');
```

Output format aliases

The $format argument of the send() and flush() should be passed as a standard mime-type string. However, for convenience and brevity Zaphpa allows indicating some simple aliases for common mime types:

```
'html' => 'text/html',
'txt' => 'text/plain',
'css' => 'text/css',
'js' => 'application/x-javascript',
'xml' => 'application/xml',
'rss' => 'application/rss+xml',
'atom' => 'application/atom+xml',
'json' => 'application/json',
```

A More Advanced Router Example

```
<?php

$router = new Zaphpa_Router();

$router->addRoute(array(
  'path'     => '/pages/{id}/{categories}/{name}/{year}',
  'handlers' => array(
    'id'         => Zaphpa_Constants::PATTERN_DIGIT, //regex
    'categories' => Zaphpa_Constants::PATTERN_ARGS,  //regex
    'name'       => Zaphpa_Constants::PATTERN_ANY,   //regex
    'year'       => 'handle_year',       //callback function
  ),
  'get'      => array('MyController', 'getPage'),
  'post'     => array('MyController', 'postPage'),
  'file'     => 'controllers/mycontroller.php'
));
```

```
// Add default 404 handler.
try {
  $router->route();
} catch (Zaphpa_InvalidPathException $ex) {
  header("Content-Type: application/json;", TRUE, 404);
  $out = array("error" => "not found");
  die(json_encode($out));
}

function handle_year($param) {
  return preg_match('~^\d{4}$~', $param) ? array(
    'ohyesdd' => $param,
    'ba' => 'booooo',
  ) : null;
}
```

Please note the "file" parameter to the `->addRouye()` call. This parameter indicates file where MyController class should be loaded from, if you do not already have the corresponding class loaded (through an auto-loader or explicit require() call).

Routing to Entities

So far we have discussed routing individual URI patterns. However, when building a RESTful API, you often need to create full Resources or Endpoints - API lingo for objects that can be managed in a full: Create, Read, Update, Delete (CRUD) lifecycle.

One way you can do this is to fully declare all four routes. But that would mean a lot of duplicated configuration. We hate code duplication, so here's a nifty shortcut you can use:

```
<?php

$router->addRoute(array(
  'path'     => '/books/{id}',
  'handlers' => array(
    'id'         => Zaphpa_Constants::PATTERN_DIGIT,
  ),
  'get'      => array('BookController', 'getBook'),
  'post'     => array('BookController', 'createBook'),
  'put'      => array('BookController', 'updateBook'),
  'delete'     => array('BookController', 'deleteBook'),
  'file'     => 'controllers/bookcontroller.php'
));
```

Pre-defined Validator Types

Zaphpa allows indicating completely custom function callbacks as validating handlers, but for convenience it also provides number of pre-defined, common validators:

```
<?php

const PATTERN_NUM        = '(?P<%s>\d+)';
```

```
const PATTERN_DIGIT      = '(?P<%s>\d+)';
const PATTERN_MD5        = '(?P<%s>[a-z0-9]{32})';
const PATTERN_ALPHA      = '(?P<%s>(?:/?[-\w]+))';
const PATTERN_ARGS       = '?(?P<%s>(?:/.+)+)';
const PATTERN_ARGS_ALPHA = '?(?P<%s>(?:/[-\w]+)+)';
const PATTERN_ANY        = '(?P<%s>(?:/?[^/]*))';
const PATTERN_WILD_CARD  = '(?P<%s>.*)';
const PATTERN_YEAR       = '(?P<%s>\d{4})';
const PATTERN_MONTH      = '(?P<%s>\d{1,2})';
const PATTERN_DAY        = '(?P<%s>\d{1,2})';
```

You may be able to guess the functionality from the regexp patterns associated with each pre-defined validator, but let's go through the expected behavior of each one of them:

- `PATTERN_NUM` - ensures a path element to be numeric
- `PATTERN_DIGIT` - alias to `PATTERN_NUM`
- `PATTERN_MD5` - ensures a path element to be valid MD5 hash
- `PATTERN_ALPHA` - ensures a path element to be valid alpha-numeric string (i.e. latin characters and numbers, as defined by \w pattern of regular expression syntax).
- `PATTERN_ARGS` - is a more sophisticated case that takes some explanation. It tries to match multiple path elements and could be useful in URLs like:
  - `/news/212424/**us/politics/elections**/some-title-goes-here/2012` where "us/politics/elections" is a part with variable number of "categories". To parse such URL you could define a validator like:

```
<?php

'path'     => '/news/{id}/{categories}/{title}/{year}',
'handlers' => array(
  'id'          => Zaphpa_Constants::PATTERN\_NUM,
  'categories'  => Zaphpa_Constants::PATTERN\_ARGS,
  'title'       => Zaphpa_Constants::PATTERN\_ALPHA,
  'year'       => Zaphpa_Constants::PATTERN\_YEAR,
 ),
```

and you would get the function arguments in the callback as:

```
<?php

[params] => Array
(
  [id] => 212424
  [categories] => Array
      (
          [0] => us
          [1] => politics
          [1] => elections
      )
  [title] => some-title-goes-here
  [year] => 2012
)
```

- `PATTERN_ARGS_ALPHA` - acts the exact same way as `PATTERN_ARGS` but limits character set to alpha-numeric ones.
- `PATTERN_ANY` (default) - matches any one argument
- `PATTERN_WILD_CARD` - "greedy" version of `PATTERN_ANY` that can match multiple arguments
- `PATTERN_YEAR` - matches a 4-digit representation of a year.
- `PATTERN_MONTH` - matches 1 or 2 digit representation of a month
- `PATTERN_DAY` - matches 1 or 2 digit representation of a numeric day.

For more custom cases, you can use a custom regex:

```
'handlers' => array(
  'id'   => Zaphpa_Constants::PATTERN_DIGIT, //numeric
  'full_date' => Zaphpa_Template::regex('\d{4}-\d{2}-\d{2}'); // custom regex
),
```

or attach a validator (you can also think of it as: URL parameter parser) callback function where you can get almost unlimited flexibility:

```
'handlers' => array(
  'id'         => Zaphpa_Constants::PATTERN_DIGIT, //numeric
  'uuid'       => 'handle_uuid',       //callback function
),
```

The output of the custom validator callback should match that of a PHP regex call i.e.: should return a parsed array of matches or a null value.

Appendix A: Setting Up Zaphpa Library

You need to register a PHP script to handle all HTTP requests. For Apache it would look something like the following:

```
RewriteEngine On
RewriteRule "(^|/)\." - [F]
RewriteCond %{DOCUMENT_ROOT}%{REQUEST_FILENAME} !-f
RewriteCond %{DOCUMENT_ROOT}%{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_URI} !=/favicon.ico
RewriteRule ^ /your_www_root/api.php [NC,NS,L]
```

Please note that this configuration is for a httpd.conf, if you are putting it into an .htaccess file, you may want to remove the leading %{DOCUMENT_ROOT} in the corresponding RewriteConds.

The very first RewriteRule is a security-hardening feature, ensuring that system files (the ones typically starting with dot) do not accidentally get exposed.

For Nginx, you need to make sure that Nginx is properly configured with PHP-FPM as CGI and the actual configuration in the virtualhost may look something like:

```
location / {
  # the main router script
```

```
  if (!-e $request_filename) {
    rewrite ^/(.*)$ /api.php?q=$1 last;
  }
}
```

That's it, for now.