APIx Documentation

Release 0.3.6

Franck Cassedanne

March 17, 2017

Contents

CHAPTER 1

The Manual

Contains a quick tour of APIx allowing to get started fast.

APIx Official Manual

Introduction

APIx is a (micro-)framework to build RESTful Web services. It will run alognside your existing framework/application with minimum fuss.

Some of its features:

- Supports **many data inputs** such as GET/POST parameters, XML, JSON, CSV, ...
- Provides **various output representation** such as XML, JSONP, HTML, PHP, ...
- Provides **on-demand resources documention**, using GET /help or 'OPTIONS'.
- Uses **annotations to document** and **set service behaviours**.
- Handles **most HTTP methods**, including PUT, DELETE, HEAD, OPTIONS and PATCH (TRACE to some extent).
- Bundled with **many plugins and adapters** including:
  - Basic HTTP Authentication
  - Digest HTTP Authentication
  - Caching through Redis, APC, Memcached, PDO, etc
  - Extensible Plugin architecture
- **Follows the standards** such as:
  - **RFC 2616** - Hypertext Transfer Protocol – HTTP/1.1
  - **RFC 2617** - HTTP Authentication: Basic and Digest Access Authentication
  - **RFC 2388** - Returning Values from Forms (multipart/form-data)
  - **RFC 2854** - The 'text/html' Media Type
  - **RFC 4627** - The application/json Media Type for JavaScript Object Notation (JSON)
  - **RFC 4329** - Scripting Media Types
  - **RFC 2046** - Multipurpose Internet Mail Extensions

  - RFC 3676 - The Text/Plain Format and DelSp Parameters
  - RFC 3023 - XML Media Types
  - etc...

- Provides **method-override** usign X-HTTP-Method-Override (Google recommendation) and/or using a query-param (customisable).
- Supports **content negotiation** (which can also be overriden).
- Take advantages of network caches – supports HEAD test.
- Available as a standalone phar file, composer, pear package, or via github.

Installation

There are several options for installing APIx. We recommend the **phar** method for optimal speed. However, APIx is also available via **composer** for easy integration into your project.

Apix requires PHP 5.3 or later.

PHAR

Download apix.phar and include it in your project like this:

```
include '/path/to/apix.phar';
$apix = new Apix\Server;
```

The apix.phar file also contains a CLI interface that can be used to self-update.

```
$ php apix.phar --selfupdate
```

Composer

Integrate APIx into your existing Composer project by adding the following to your composer.json file:

```
{
  "require": {
    "apix/apix": "0.3.*"
  }
}
```

```
include "vendor/autoload.php";
$apix = new Apix\Server;
```

Quick Start

The most basic example creates and serves a route that echos the content passed in through the URL parameter called *name*. This route would be accessed through http://www.example.com/hello/myname and would return 'Hello myname'.

```
try {
    // Instantiate the server (using the default config)
    $api = new Apix\Server(require 'config.php');
```

```
    // Create a GET handler $name is required
    $api->onRead('/hello/:name', function($name) {
        return array('Hello, ' . $name);
    });

    $api->run();

} catch (\Exception $e) {
    header($_SERVER['SERVER_PROTOCOL'] . ' 500 Internal Server Error', true, 500);
    die("<h1>500 Internal Server Error</h1>" . $e->getMessage());
}
```

Another example using annotations.

```
try {
    // Instantiate the server (using the default config)
    $api = new Apix\Server(require 'config.php');

    // $type and $stuff are required parameters.
    // $optional is not mandatory.
    $api->onRead('/search/:type/with/:stuff/:optional',
        /**
         * Search for things by type that have stuff.
         *
         * @param     string  $type         A type of thing to search upon
         * @param     string  $stuff        One or many stuff to filter against
         * @param     string  $optional     An optional field
         * @return    array
         * @api_auth  groups=clients,employes,admins users=franck,jon
         * @api_cache ttl=12mins tags=searches,indexes
         */
        function($type, $stuff, $optional = null) {
            // some logic
            return $results;
        }
    );

    $api->run();

} catch (\Exception $e) {
    header($_SERVER['SERVER_PROTOCOL'] . ' 500 Internal Server Error', true, 500);
    die("<h1>500 Internal Server Error</h1>" . $e->getMessage());
}
```

config.php

The following example configuration file is used in the above examples. Details on the function of these options may be found in the `/config` documentation.

```
<?php
namespace Apix;

$c = array(
    'api_version'       => '0.1.0.empty-dumpty',
    'api_realm'         => 'api.domain.tld',
    'output_rootNode'   => 'apix',
    'input_formats'     => array('post', 'json', 'xml'),
```

```
    'routing'           => array(
        'path_prefix'       => '/^(\/\w+\.\w+)?(\/api)?\/v(\d+)/i',
        'formats'           => array('json', 'xml', 'jsonp', 'html', 'php'),
        'default_format'    => 'json',
        'http_accept'       => true,
        'controller_ext'    => true,
        'format_override'   => isset($_REQUEST['_format'])
                                ? $_REQUEST['_format']
                                : false,
    )

);

// Resources definitions
$c['resources'] = array(
    '/help/:path' => array(
        'redirect' => 'OPTIONS'
    ),
    '/*' => array(
        'redirect' => 'OPTIONS',
    )
);

// Service definitions
$c['services'] = array(

    // Auth examples (see plugins definition)
    'auth_example' => function() use ($c) {
        $adapter = new Plugin\Auth\Basic($c['api_realm']);
        $adapter->setToken(function(array $current) use ($c) {
            $users = Service::get('users_example');
            foreach ($users as $user) {
                if ($current['username'] == $user['username'] && $current['password'] == $user['api_
                    Service::get('session', $user);
                    return true;
                }
            }

            return false;
        });
        return $adapter;
    },

    // This is used by the auth_example service defined above.
    'users_example' => function() {
        return array(
            0 => array(
                'username' => 'myuser', 'password' => 'mypass', 'api_key' => '12345', 'group' => 'adm
            )
        );
    },

    // This is used by the auth_example service defined further above.
    'session' => function($user) {
        $session = new Session($user['username'], $user['group']);
        if (isset($user['ips'])) {
            $session->setTrustedIps((array) $user['ips']);
        }
```

```
        $session->addData('api_key', $user['api_key']);
        Service::set('session', $session);
    }

);

// Plugins definitions
$c['plugins'] = array(
    'Apix\Plugin\OutputSign',
    'Apix\Plugin\OutputDebug' => array('enable' => DEBUG),
    'Apix\Plugin\Tidy',
    'Apix\Plugin\Auth' => array('adapter' => $c['services']['auth_example']),
);

// Init is an associative array of specific PHP directives. They are
// recommended settings for most generic REST API server and should be set
// as required. There is most probably a performance penalty setting most of
// these at runtime so it is recommneded that most of these (if not all) be
// set directly in PHP.ini/vhost file on productions servers -- and then
// commented out. TODO: comparaison benchmark!?
$c['init'] = array(
    'display_errors'            => DEBUG,
    'init_log_errors'           => true,
    'error_log'                 => '/tmp/apix-server-errors.log',
);

$c['default'] = array(
    'services' => array(),
    'resources' => array(
        'OPTIONS' => array(
            'controller' => array(
                'name' => __NAMESPACE__ . '\Resource\Help',
                'args' => null
            ),
        ),
        'HEAD' => array(
            'controller' => array(
                'name' => __NAMESPACE__ . '\Resource\Test',
                'args' => null
            ),
        ),
    )
);
$c['config_path'] = __DIR__;
return $c;
```

Configuration Options

Available Options

Do not make changes to the distributed configuration file. Rather, include the distributed configuration file in your own configuration file and overwrite the items that you would like to. This eases the upgrade path when defaults are changed and new features are added.

The configuration variable is an associative array containing the following keys and values:

api_version

```
$config['api_version'] => '0.1.0.empty-dumpty'
```

The API version string allowing a userbase to keep track of API changes. It is definied as major.minor.maintenance[.build] where:

Major Increase for each changes that may affect or not be compatible with a previous version of the API.

Bumping the major generally imply a fresh new production deployment so the previous version can (and should) be left intact for those that depend upon it.

Minor Increases each time there are new addition e.g. a new resource.

Maintenance Increases each time there are modifications to existing resource entities and which don't break exisiting definitions.

Build Can be use for arbitrary naming such as 0.1.0.beta, 0.1.1.rc3 (third release candidate), 0.1.2.smoking-puma, 0.1.30.testing

api_realm

```
$config['api_realm'] => 'api.example.com'
```

The API realm name. Used in few places, most notably as part of the version string in the header response. It is also used as part of some authentication mechanisms e.g. Basic and Digest. Should always be a generic/static string and cannot be used to define server instance. In other words, DO NOT use $_SERVER['SERVER_NAME'] to set this option!

output_rootNode

```
$config['output_rootNode'] => 'apix'
```

Define the name of the data output topmost node which contains the various nodes generated by the response output. The signature and debug nodes are also contained within this node if they are enabled in the plugins section.

input_formats

```
$config['input_formats'] => array('post', 'json', 'xml')
```

The array of available data formats for input representation:

POST Body post data

JSON Light text-based open standard designed for human-readable data interchange.

XML Generic and standard markup language as defined by XML 1.0 schema.

Note that at this stage only UTF-8 is supported.

routing

The routing value is an associative array with the following keys: path_prefix, formats, default_format, http_accept, controller_ext, and format_override.

```
$config['routing'] => array(
    'path_prefix' => ...,
    'formats' => ...,
    'default_format' => ...,
    'http_accept' => ...,
    'controller_ext' => ...,
    'format_override' => ...
);
```

path_prefix

```
'path_prefix' => '/-(\/\w+\.\w+)?(\/api)?\/v(\d+)/i'
```

The regular expression representing the path prefix from the Request-URI. Allows the server to retrieve the path without the route prefix, handling variation in version numbering, Apache's mod_rewrite, nginx location definitions, etc...

Should match '/index.php/api/v1/entity/name?whatver...' which using mod_rewrite could then translate into 'http://www.example.com/v1/entity/name?whatver...'.

formats

```
'formats' => array('json', 'xml', 'jsonp', 'html', 'php')
```

The array of available data formats for output representation:

JSON Light text-based open standard designed for human-readable data interchange.

XML Generic and standard markup language as defined by the XML 1.0 specification. Again, other schema could be implemented if required.

JSONP Output JSON embeded within a javascript callback. Javascript clients can set the callback name using the GET/POST variable named 'callback' or default to the 'output_rootNode' value set above.

HTML Output an HTML bulleted list.

PHP Does not currently serialize the data as one would expect but just dumps the output array for now.

default_format

```
'default_format' => 'json'
```

Set the defaut output format to either JSON or XML. Note that JSON encoding is by definition UTF-8 only. If a specific encoding is required then XML should be used as the default format. In most case, JSON is favored.

http_accept

```
'http_accept' => true
```

Whether to enable the negotiation of output format from an HTTP Accept header. This is the expected and most RESTful way to set the output format. See **RFC 2616** for more information.

controller_ext

```
'controller_ext' => true
```

Whether to allow the output format to be set from the Request-URI using a file extension such as '/controller.json/id'. This is handy and common practice but fairly un-RESTful. The extension overrides the http_accept negotiation.

format_override

```
'format_override' => isset($_REQUEST['_format']) ? $_REQUEST['_format'] : false
```

Forces the output format to the string provided and overrides the format negotiation process. Set to false to disable. Can be use to set the format from a request parameter, or any other arbitrary methods, etc... Using `$_REQUEST` is considered un-RESTful but also can be handy in many cases e.g. forms handling.

resources

A resource definition is made of a 'Route path' (with or without named variable) pointing to a controller which may be defined as closure/lambda definitions (à la Sinatra) allowing fast prototyping, class definitions allowing for a tradition Model + Controller layout, or a redirect.

Class Definitions

```
$config['resources'] += array(
    '/hello/:name' => array(
        'controller' => array(
            'name' =>   'MyControllers\Hello', // a namespace\classname as a string
            'args' =>   array('classArg1'=>'value1', 'classArg2'=>'string') // a __constructor varia
        )
    ),
    ...
)
```

The default values to the 'resources' key set up API documentation links and should not be overwritten.

Redirects

```
$config['resources'] += array(
    '/redirect/me' => array(
        'redirect' => '/hello/world'
    ),
    ...
)
```

Perform a redirect on the path '/redirect/me' to 'hello/world'.

services

The service defintions array is mostly used as a convenient container to define some generic/shared code. For example, Authorization adapters and session data can be stored in the services array. These items can later be retrieved using Apix\Service::get().

Authentication Service Example

An example Authentication service might look something like this:

```
$config['services'] => array(
    // $config is the current configuration array
    'auth' => function() use ($config) {
        // Example implementing Plugin\Auth\Basic
        // The Basic Authentification mechanism is generally used with SSL.
        $adapter = new Apix\Plugin\Auth\Basic($config['api_realm']);
        $adapter->setToken(function(array $current) {
            $users = array(
                array('username'=>'example', 'password'=>'mypassword', group=>'admin', 'realm'=>'www
            );
            foreach ($users as $user) {
                if ($current['username'] == $user['username'] && $current['password'] == $user['passw
                    Service::get('session', $user);
                    return true;
                }
            }
            return false;
        });
        return $adapter;
    },

    // create a session object that we can use in the auth service
    'session' => function($user) {
        // Set that way solely to avoid duplicating code in auth_example.
        $session = new Session($user['username'], $user['group']);
        // Overwrite this service container, with the new Session object!
        // Apix\Plugin\Auth expects this session container to hold Apix\Session.
        Service::set('session', $session);
    }
);
```

In this example, we have both a *session* service and an *auth* service. The *auth* service makes use of the *session* service, as the session is used in other code in APIx. Another service might have been created to store or dynamically retrieve a users array.

plugins

Please see the *Plugin documentation* for more information on available event hooks and interface for Plugins.

Plugins is an associative array where each plugin is definied using the plugins class name as the key, and an array defining options for that plugin as the value. The options array is passed into the constructor for the specified plugin class. For example:

```
$config['plugins'] => array(
    'MyProject\Plugins\MyPlugin' => array('enable'=>true, 'myoption'=>'hello world')
);
```

The above code would create a new `MyProject\Plugins\MyPlugin` like this:

```
$plugin = new \MyProject\Plugins\MyPlugin(array('enable'=>true, 'myoption'=>'hello world'));
```

Currently available plugins include the following:

Output Signature

Adds the entity signature as part of the response body.

```
$config['plugins']['Apix\\Plugin\\OutputSign'] = array();
```

Output Debugging

Add some debugging information within the response body. This should be set to false in production and does have an impact on cachability.

```
$config['plugins']['Apix\\Plugin\\OutputDebug'] = array();
```

Tidy (Pretty-print)

Validates, corrects, and pretty-prints XML and HTML outputs. Various options are available. See the Tidy quickref for more information on available options.

```
$config['plugins']['Apix\\Plugin\\Tidy'] = array('indent-spaces' => 4, 'indent' => true);
```

Authentication

The authentication plugin is enabled through method/closure annotation. The following example instructs the authentication plugin allow access to the following GET resource if a user can authenticate to either the "admin" or "default" user groups.

```
/**
 * My Method Annotation
 * @api_auth  groups=admin,default  users=usera,userb
 **/
public function onRead() {
    ...
}
```

The configuration block must provide an adapter object which implements `Apix\Plugin\\Auth\\Adapter`. An *Authentication Service Example* which provides an authentication adapter is included in the *Services* section.

```
$config['plugins']['Apix\\Plugin\\Auth'] = array('adapter' => $c['services']['auth']);
```

Entity Caching

The cache plugin allows you to easily cache the output from a controller request. The full Request-URI acts as the unique cache id for a particular resource. Like the authorization plugin, this is enabled through method/closure annotation. For example:

```
/**
 * My Method Annotation
 * @api_cache  ttl=1hours  tags=tag1,tag2  flush=tag3,tag4
 **/
public function onRead() {
    ...
}
```

`Apix\Cache` is available at https://github.com/frqnck/apix-cache.

The options available for the cache plugin include an "enable" key and an "adapter" key, which requires an object implementing an `Apix\Cache\Adapter` interface.

```
$config['plugins']['Apix\Plugin\Cache'] = array(
    'enable'  => true,
    'adapter' => new Apix\Cache\APC
);
```

You could also add the caching adapter as a *services* to reuse the same cache connection throughout your project. In that case, instead of instantiating a new `Apix\Cache\APC` in your plugin configuration, you would create a service that exposes the adapter, and use that. For example:

```
$config['services']['cache'] = new Apix\Cache\APC;

$config['plugins']['Apix\Plugin\Cache'] = array(
    'enable' => true,
    'adapter' => $config['services']['cache']
);
```

init

Init is an associative array of specific PHP directives. They are recommended settings for most generic REST API servers and should be set as required. There is most probably a performance penalty setting most of these at runtime so it is recommneded that most, if not all, of these be set directly in php.ini/vhost files on productions servers and then commented out. Values included here will overwrite the values provided in php.ini or other PHP init files.

display_errors

```
'display_errors' => true
```

Whether to display errors or not. This should be set to false in production.

init_log_errors

```
'init_log_errors' => true
```

Enable or disable php error logging.

error_log

```
'error_log' => '/path/to/error.log'
```

Path to the error log file.

html_errors

```
'html_errors' => true
```

Enable or disable html_errors.

zlib.output_compression

```
'zlib.output_compression' => true
```

Whether to transparently compress outputs using GZIP. If enabled, this options will also add a 'Vary: Accept-Encoding' header to response objects.

memory_limit

```
'memory_limit' => '64M'
```

Maximum amount of memory a script may consume.

max_execution_time

```
'max_execution_time' => 15
```

The timeout in seconds. Be aware that web servers such as Apache also have their own timout settings that may interfere with this. See your web server manual for specific details.

post_max_size

```
'post_max_size' => '8M'
```

Maximum size of POST data that this script will accept. Its value may be 0 to disable the limit.

max_input_time

```
'max_input_time' => 30
```

Maximum amount of time each script may spend parsing request data.

max_input_vars

```
'max_input_vars' => 100
```

Maximum number of GET/POST input variables.

max_input_nesting_level

```
'max_input_nesting_level' => 64
```

Maximum input variable nesting level.

variables_order

```
'variables_order' => 'GPS'
```

Determines which super global are registered and in which order these variables are then populated. G,P,C,E & S are abbreviations for the following respective super globals: GET, POST, COOKIE, ENV and SERVER. There is a performance penalty paid for the registration of these arrays and because ENV is not as commonly used as the others, ENV is not recommended on productions servers. You can still get access to the environment variables through getenv() should you need to.

request_order

```
'request_order' => 'GP'
```

This directive determines which super global data (G,P,C,E & S) should be registered into the super global array REQUEST. If so, it also determines the order in which that data is registered. The values for this directive are specified in the same manner as the variables_order directive, EXCEPT one. Leaving this value empty will cause PHP to use the value set in the variables_order directive. It does not mean it will leave the super globals array REQUEST empty.

Example Project

Project Layout

Assume that our project is laid out as follows:

```
MyProject/
|-- composer.json
|-- config/
|   |-- config.php
|   |-- credentials.php
|   |-- plugins.php
|   |-- resources.php
|   `-- services.php
|-- controllers/
|   |-- Goodbye.php
|   `-- Hello.php
|-- models/
|-- public/
|   |-- .htaccess
|   `-- index.php
`-- vendor/
    |-- apix/
    |   |-- apix/
    |   `-- cache/
    |-- autoload.php
    `-- composer/
```

For the sake of this example, we'll put `MyProject` directly in our webroot. In most environments, you will want to expose **only** the `public` directory. Download `MyProject here`.

composer.json

We use Composer to pull in the required external libraries, including the APIx framework and the APIxCache library. A `composer.json` file for our project might look something like this (assuming you layout your controllers and

models using the PSR-4 specification:

```
{
    "name":"myproject/myproject",
    "require": {
        "apix/apix": "0.3.*",
        "apix/cache": "1.1.*"
    },
    "autoload": {
        "psr-4":{
            "MyProject\\Controllers\\": "controllers/",
            "MyProject\\Models\\": "models/"
        }
    }
}
```

Configuration

Lets first look at what a sample `./config/config.php` file might look like. Bear in mind that this is an example, and none of these extra configuration files are actually necessary. You could easily edit everything in a single file. Then we'll look at each of the required configuration files that help us define our RESTful API.

```
<?php

define('DEBUG', true);

// Set our configuration variable to the default value
$config = require "../vendor/apix/apix/src/data/distribution/config.dist.php";
$config['api_version']     = '0.0.1.spamandeggs';
$config['api_realm']       = 'api.myproject.com';
$config['output_rootNode'] = 'myproject';

// We're testing this using Apache with no virtual hosts - so we'll have to redefine
// the routing path_prefix
$config['routing']['path_prefix'] = '/^\/MyProject\/public\/v(\d+)/i';

// Include credentials that we can use elsewhere in custom defined services, etc.
$config['credentials']     = require 'credentials.php';

// Include the resources we have defined in our resources.php configuration file
$config['resources']      += require 'resources.php';

// Include the services we have defined in our services.php configuration file.
// If a service is redefined in the services.php file, use that instead.
$config['services']       = array_merge($config['services'], require 'services.php');

// Include the plugins we have defined in our plugins.php configuration file
$config['plugins']        = array_merge($config['plugins'], require 'plugins.php');

return $config;
```

config/credentials.php

The credentials file is used to store any credentials used to make connections to an outside data source. For example, you might store information about your caching server or database connections.

```
<?php

return array(
    // use a Redis instance for caching
    'redis' => array(
        'servers' => array(
            array('127.0.0.1', 6379)
        ),
        'options' => array(
            'atomicity' => false,
            'serializer' => 'php'
        )
    )
);
```

config/resources.php

The resources file is where we'll store information about all of our available routes. We'll be using class based controllers in this example. If we wanted to use closures, we could define these as lambda functions.

```
<?php

return array(
    '/hello/:name' => array(
        'controller' => array(
            'name' => 'MyProject\Controllers\Hello',
            'args' => null
        )
    ),
    '/goodbye/:name' => array(
        'controller' => array(
            'name' => 'MyProject\Controllers\Goodbye',
            'args' => null
        )
    )
);
```

We've now defined two routes that we'll be able to access at http://api.example.com/v1/hello/:name and http://api.example.com/v1/goodbye/:name. The HTTP Method (**RFC 2616**) available for these functions will be defined directly in the controllers themselves.

config/services.php

We define a caching adapter which can be used through the project as a whole, and also by the caching plugin to allow for easy caching of output content. If you include this service while trying out this example, you **will** have to set up a Redis instance. If you'd prefer to skip this, simply return an empty array both here and in the plugins configuration file.

```
<?php

use Apix\Cache;
use Apix\Service;

return array(
    // we'll reference the existing $config variable to retrieve our redis credentials
```

```
    'cache' => function() use ($config) {
        $redis = new \Redis();
        foreach($config['credentials']['redis']['servers'] as $redis_server) {
            $redis->connect($redis_server[0], $redis_server[1]);
        }
        $adapter = new Cache\Redis($redis, $config['credentials']['redis']['options']);

        // Reset this service definition so that continuous calls do not recreate a new adapter
        // but simply return the existing one.
        Service::set('cache', $adapter);
        return $adapter;
    }
);
```

config/plugins.php

We can define our own plugins if we choose. Lets add in caching capabilities, which are not turned on in the default conguration. We'll be relying on the Apix\Cache library to provide the caching adapter. The caching adpater will be defined in the services configuration file. This example also assumes that the services configuration file has already been processed, as it makes use of the cache service defined there.

```
<?php

return array(
    // Plugin to cache the output of the controllers. The full Request-URI acts as
    // the unique cache id.  Caching is enabled through a controller method or closure's
    // annotation
    // e.g. * @api_cache  ttl=5mins  tags=tag1,tag2  flush=tag3,tag4
    'Apix\Plugin\Cache' => array('enable'=>false, 'adapter'=>$config['services']['cache'])
);
```

Controllers

We've defined two resources above that each point to separate controller classes.

controllers/Goodbye.php

The following controller will define a `GET` resource.

```
<?php

namespace MyProject\Controllers;
use Apix\Request;
use Apix\Response;

/**
 * Goodbye
 *
 * Lets say goodbye to people nicely.
 *
 * @api_public  true
 * @api_version 1.0
 * @api_auth    groups=public
 */
```

```
class Goodbye {

    /**
     * Goodbye
     *
     * Say Goodbye
     *
     * @param      string     $name        Who should we say goodbye to?
     * @return     array
     * @api_cache  ttl=60sec  tag=goodbye  Cache this call for 60 seconds
     */
    public function onRead(Request $request, $name) {
        if(strlen(trim($name)) == 0) {
            throw new \Exception("I don't know who I'm saying goodbye to!");
        }

        return array("goodbye" => "goodbye, " . trim($name));
    }
}
```

controllers/Hello.php

The following controller will define both `GET` and `POST` resources. Other methods could also be defined here using the typical **CRUD** methods.

```
<?php

namespace MyProject\Controllers;
use Apix\Request;
use Apix\Response;

/**
 * Hello
 *
 * Lets say hello to people nicely.
 *
 * @api_public  true
 * @api_version 1.0
 * @api_auth    groups=public
 */
class Hello {

    /**
     * Hello
     *
     * Say Hello to someone
     *
     * @param      string     $name        Who should we say hello to?
     * @return     array
     * @api_cache  ttl=60sec  tag=goodbye  Cache this call for 60 seconds
     */
    public function onRead(Request $request, $name) {
        if(strlen(trim($name)) == 0) {
            // Return a 400 if they didn't pass in a name
            throw new \Exception("I don't know who I'm saying hello to!", 400);
        }
```

```
        return array("greeting" => "hello, " . trim($name));
    }

    /**
     * Hello
     *
     * Say hello to someone using the POSTED greeting.
     *
     * @param      string     $name        Who should we say hello to?
     * @param      string     $greeting    How should we say hello?
     * @return     array
     * @api_cache  ttl=60sec  tag=goodbye  Cache this call for 60 seconds
     */
    public function onCreate(Request $request, $name) {
        if(strlen(trim($name)) == 0) {
            // Return a 400 if they didn't pass in a name
            throw new \Exception("I don't know who I'm saying hello to!", 400);
        }

        $data = $request->getBodyData();
        if($data == null || !is_array($data)) {
            // Return a 400 if they didn't pass in any POST data
            throw new \Exception("Could not read the POST request body", 400);
        }
        $greeting = array_key_exists('greeting', $data) ? (string) $data['greeting'] : "hello";

        return array("greeting" => $greeting . ', ' . trim($name));
    }
}
```

public/index.php

In this example, all calls to our API will be directed through the main index file. By exposing only the `public` directory via our webserver, we can effectively protect the other content in our project tree. This helps to avoid security leaks caused by the accidental presence of a temporary swap file or leftover text file that might leak confidential information.

```
<?php

require_once '../vendor/autoload.php';

try {

    $api = new Apix\Server(require '../config/config.php');
    echo $api->run();
} catch (\Exception $e) {
    header($_SERVER['SERVER_PROTOCOL'] . ' 500 Internal Server Error', true, 500);
    die("<h1>500 Internal Server Error</h1>" . $e->getMessage());
}
```

public/.htaccess

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
```

```
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

Try it out

When all is appropriately setup, access the following URL to access self-generated documentation:

`curl http://localhost/MyProject/public/v1/help?_format=json`

You should see something like the following:


```
{
    "myproject": {
        "debug": {
            "headers": {
                "Vary": "Accept"
            },
            "memory": "1.18 MB~1.2 MB",
            "output_format": "json",
            "request": "GET /MyProject/public/v1/help HTTP/1.1",
            "router_params": [
                "help"
            ],
            "timestamp": "Thu, 13 Mar 2014 21:32:19 GMT",
            "timing": "0.018 seconds"
        },
        "help": {
            "items": [
                {
                    "api_auth": "groups=public",
                    "api_public": "true",
                    "api_version": "1.0",
                    "description": "Lets say hello to people nicely.",
                    "methods": {
                        "GET": {
                            "api_cache": "ttl=60sec  tag=goodbye  Cache this call for 60 seconds",
                            "description": "Say Hello to someone",
                            "params": {
                                "name": {
                                    "description": "Who should we say hello to?",
                                    "name": "name",
                                    "required": true,
                                    "type": "string"
                                }
                            },
                            "return": "array",
                            "title": "Hello"
                        },
                        "POST": {
                            "api_cache": "ttl=60sec  tag=goodbye  Cache this call for 60 seconds",
                            "description": "Say hello to someone using the POSTED greeting.",
                            "params": {
                                "greeting": {
                                    "description": "How should we say hello?",
                                    "name": "greeting",
                                    "required": false,
                                    "type": "string"
```

```
                    },
                    "name": {
                        "description": "Who should we say hello to?",
                        "name": "name",
                        "required": true,
                        "type": "string"
                    }
                },
                "return": "array",
                "title": "Hello"
            }
        },
        "path": "/hello/:name",
        "title": "Hello"
    },
    {
        "api_auth": "groups=public",
        "api_public": "true",
        "api_version": "1.0",
        "description": "Lets say goodbye to people nicely.",
        "methods": {
            "GET": {
                "api_cache": "ttl=60sec  tag=goodbye  Cache this call for 60 seconds",
                "description": "Say Goodbye",
                "params": {
                    "name": {
                        "description": "Who should we say goodbye to?",
                        "name": "name",
                        "required": true,
                        "type": "string"
                    }
                },
                "return": "array",
                "title": "Goodbye"
            }
        },
        "path": "/goodbye/:name",
        "title": "Goodbye"
    },
    {
        "description": "This resource entity provides in-line referencial to all the API
        "methods": {
            "GET": {
                "description": "This resource entity provides in-line referencial to all
                "example": "<pre>GET /help/path/to/entity</pre>",
                "id": "help",
                "params": {
                    "filters": {
                        "description": "Filters can be use to narrow down the resultset.'
                        "name": "filters",
                        "required": false,
                        "type": "array"
                    },
                    "path": {
                        "description": "A string of characters used to identify a resourc
                        "name": "path",
                        "required": false,
                        "type": "string"
```

```
                                }
                            },
                            "see": "<pre>http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.2
                            "title": "Display the manual of a resource entity",
                            "usage": "The OPTIONS method represents a request for information about t
                        },
                        "OPTIONS": {
                            "api_link": [
                                "OPTIONS /path/to/entity",
                                "OPTIONS /"
                            ],
                            "description": "The OPTIONS method represents a request for information a
                            "params": {
                                "filters": {
                                    "description": "An array of filters.",
                                    "name": "filters",
                                    "required": false,
                                    "type": "array"
                                },
                                "server": {
                                    "description": "The main server object.",
                                    "name": "server",
                                    "required": true,
                                    "type": "Server"
                                }
                            },
                            "private": "1",
                            "return": "array  The array documentation.",
                            "title": "Outputs info for a resource entity."
                        }
                    },
                    "path": "OPTIONS",
                    "title": "Help"
                },
                {
                    "description": "",
                    "methods": {
                        "HEAD": {
                            "cacheable": "true",
                            "codeCoverageIgnore": "",
                            "description": "The HEAD method is identical to GET except that the serve
                            "link": "http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.4",
                            "return": "null",
                            "title": "HTTP HEAD: test action handler"
                        }
                    },
                    "path": "HEAD",
                    "title": null
                }
            ]
        },
        "signature": {
            "client_ip": "127.0.0.1",
            "resource": "GET ",
            "status": "200 OK - successful"
        }
    }
}
```

Test out POSTing to the `/hello/:name` resource using curl.

```
curl -X POST -d "greeting=hola" http://localhost/MyProject/public/v1/hello/world?_format=j
```


```
{
    "myproject": {
        "debug": {
            "headers": {
                "Vary": "Accept"
            },
            "memory": "1.14 MB~1.15 MB",
            "output_format": "json",
            "request": "POST /MyProject/public/v1/hello/world?_format=json HTTP/1.1",
            "router_params": {
                "name": "world"
            },
            "timestamp": "Thu, 13 Mar 2014 21:33:19 GMT",
            "timing": "0.02 seconds"
        },
        "hello": {
            "greeting": "hola, world"
        },
        "signature": {
            "client_ip": "127.0.0.1",
            "resource": "POST /hello/:name",
            "status": "200 OK - successful"
        }
    }
}
```


Plugins

An introduction

The plugins blah blahh...

Installation

You can add plugins blah blahh...

Managing plugins

You can add plugins blah blahh...

ManPage `Apix\Plugin\ManPage`

The ManPage Plugin for APIx is just like man page for UNIX.

An intro

Its main goal is to generate **up-to-date** and **human-friendly** manual of your API and document all of its resources.

The manual comes in two parts:

- **The main section which provides:**
  - a list of all your (public) resources.
  - documents all the Inputs and Outputs.
- **The resource section which:**
  - describes each resource in details,
  - uses PHPDoc (and Javadoc) annotation format.

Note: This plugin is part of the core distribution and enable by default. For more information see *Managing plugins*.

Tip: You can learn much more about the routing system in the `Routing chapter`.

Plugin options

Option	Description
enable *boolean*	Default True. Wether to enable or not this plugin.
example *string*	String to append to extracted @example.
see *string*	String to append to extracted @see.
link *string*	String to append to extracted @link.
copyright *string*	String to append to extracted @copyright.
license *string*	String to append to extracted @license.

Basic usage

Here is an example of annotating a simple resource.

```
$api = new Apix\Server();

$api->onRead('/beers/from/:where/:type',
    /**
     * Title - short description.
     * Long description.
     * @param   string      $where      Default "London". The location name.
     * @param   type        $type       Optional. A beer type e.g. Larger, Pale Ale, Stout.
     * @global  integer     $limit      Default 100. The number of brand to list.
     * @return type                     The response description.
     */
     function ($where, $type=null) {
        // ...
     }
);
```

Generic annotation

Title - Short description A short and concise description or title for the resource. It represents its **Synopsis**. It cannot be more than one line long. And generally ends with a period. Markup of any kind are not permitted.

Long description A more detailled description of the resource which can pan over multi-lines. Markup are permitted.

Essential annotation tags

Annontation	Description
@param *type $name description*	Documents a resource path parameter.
@global *type $name description*	Documents a request's query string parameter.
@return *type description*	Provides the **Response** of a resource.

Note: If the @param or @global are optional with a default value. It is good practice to indicate it before the description as per the example further above.

Additional tags

Annontation	Description
@usage */resource/:var*	Overrides the Synopsy's Usage field.
@example *description*	This tag is use to
@see *reference*	Provides a "See also" to some element of documentation.
@link *<url>*	Displays a link to some pages.
@copyright *description*	Documents some copyrights details.
@license *<url> name*	Provides the name and URL of a relevant license.

Auth `Apix\Plugin\Auth`

The Auth Plugin for APIx provides an authentification mechanism for your API.

This is enable thru method/closure's annotation for each resource that require users or groups authentification.

Note: This plugin is part of the core distribution. It is enable by default. For more information see *Managing plugins*.

Generic Usage

In order to inforce authentification of a resource, your need to use the `@api_auth` tag annotation.

```
use Apix;

$api = new Apix\Server();

$api->onRead('/resource/:name',
    /**
     * ...
     *
     * @api_auth   groups=clients,testers   users=franck
     */
     function ($name) {
        // whatever...
     }
);
```

In the example above, only the specified *users* and/or *groups* will be given access on succesful autentification.

Authentification is skipped altogether, if:

- *groups* and *users* are not provided in `@api_auth`,
- if one of the *group* is set to the same value as 'public_group' (set to "public" by default).

Configurable options

The main configurable options are:

public_group This options is set to the string **public** by default. Use this option to change this value to fit your requirments.

adapter Currently, two adapters are shipped with the main distribution.

Apix\Plugin\Auth\Basic This adapter provides HTTP Basic access authentication.

Use this over HTTPS as the credentials are not encrypted or hashed in any way.

An example of implementation:

```
use Apix;
...
$adapter = new Plugin\Auth\Basic($c['api_realm']);
$adapter->setToken(function (array $current) use ($c) {
    $users = Service::get('users_example');
    foreach ($users as $user) {
            if (
                $current['username'] == $user['username']
                && $current['password'] == $user['api_key']
            ) {
                Service::get('session', $user);

                return true;
            }
        }

        return false;
    });
```

Apix\Plugin\Auth\Digest This adapter provides HTTP Digest access authentication.

It is use to encrypt and salt the user's credentials can be used with or without SSL.

An example of implementation:

```
use Apix;
...
$adapter = new Plugin\Auth\Digest($c['api_realm']);
$adapter->setToken(function (array $current) use ($c) {
    $users = Service::get('users_example');
    foreach ($users as $user) {
        if (
            $user['username'] == $current['username']
            && $user['realm'] == $c['api_realm']
        ) {
            Service::get('session', $user);

            // Digest match againt your selected token.
            return $user['api_key'];
        }
    }
```

```
    return false;
});
```

Cache `Apix\Plugin\Cache`

The Cache Plugin for APIx provides a caching layer for your APi's resources.

Cachign is set either:

- globaly using the configurable options,
- and/or per individual resource using the `@api_cache` tag annotation.

Global usage

Caching can be set or not globally. A set of main configurable options are avilabale.

These options also act as the default and inherited option for the annotated subtags.

Configurable options

enable This options is set to the **True** by default. (TODO: review this part of the code)

adapter By default, this option is set to **''ApixCacheApc''** which handles APC and APCu.

There are adapters for Redis, Memcached and many other data/cache stores.

Checkout Apix Cache for a list of available adapters and their specific options.

default_ttl This options is set to **10mins** by default. This is the cache lifetime, `null` stands forever.

runtime_flush This options is set to **True** by default.

If set to True the given tags are flushed at runtime. It is recommened to disable this and do this offline instead for instance using Cronjob.

append_tags This holds an array of tags that will be appended to items being cached.

For instance, you may want to use this option to automatically tag all new entries with the major version of your API 'v1', or with 'dev' while developing.

Usage thru annotation

Tag: `@api_cache` Generally, to caches the output of a resource transparently you will use a method/closure's annotation using the `@api_cache` tag and configure it using some subtags. The *Request-URI* acts as the unique cache id.

For instance, to cache a resource for 45 minutes you would annotate it as follow:

```
use Apix;

$api = new Apix\Server();

$api->onRead('/resource/:name',
    /**
     * ...
     *
     * @api_cache   ttl=45mins   tags=tag1,tag2
     */
     function ($name) {
        // whatever...
     }
);
```

In this example, the resource will be cached under the cache id */resource/whatever* (the Request-URI) for the amount of time provided in the subtag `ttl`. Susequent requests to */resource/whatever* will hit the cache or regenerate it if it is empty (or expired).

Additionally, this resource will be cached with the given tags which can be useful for grouping cached data together. For instance, this allow to run periodic purging of some specific tags. It is also useful when grouping data to unique user-group, client-group, use-cases, etc...

Subtags

ttl This sets the time-to-live in seconds. It takes either an integer representating a value in seconds or a textual datetime string such as "10days". `ttl=null` means *forever*. If not set, it will use the value from 'default_ttl'.

tags If provided, the cached entry will be tagged with the given tag(s).

flush If 'runtime_flush' is enable in th configurable options then the given

Additionally, if a'runtime_flush' is st, some tags maybe flushed/purged from the cache.

Make your own

APIx comes prepackaged with several plugins, but there is nothing stopping you from extending this functionality and building your own. There are two available builtin plugin types that you may extend which implement an Observer patter using SplObserver. The basic plugin architecture is relatively straightforward but we'll delve specifically into the *Entity Plugins* below.

Plugins are activated by "hook" calls throughout the APIx process and should implement the SplObserver::update method. The plugin hook is defined by a static variable of the same name in your Plugin class.

Here is an example taken from the `OutputDebug` class.

```
<?php

namespace Apix\Plugin;
use Apix\Response;

class OutputDebug extends PluginAbstract {

    public static $hook = array('response', 'early');
```

```
    protected $options = array(
        'enable'     => false,              // whether to enable or not
        'name'       => 'debug',            // the header name
        'timestamp'  => 'D, d M Y H:i:s T', // stamp format, default to RFC1123
        'extras'     => null,               // extras to inject, string or array
    );

    public function update(\SplSubject $response) {
        if (false === $this->options['enable']) {
            return false;
        }

        $request = $response->getRequest();
        $route = $response->getRoute();
        $headers = $response->getHeaders();
        $data = array(
            'timestamp'     => gmdate($this->options['timestamp']),
            'request'       => sprintf('%s %s%s',
                                    $request->getMethod(),
                                    $request->getRequestUri(),
                                    isset($_SERVER['SERVER_PROTOCOL'])
                                    ? ' ' . $_SERVER['SERVER_PROTOCOL'] : null
                                ),
            'headers'       => $headers,
            'output_format' => $response->getFormat(),
            'router_params' => $route->getParams(),
        );

        if (defined('APIX_START_TIME')) {
            $data['timing'] = round(microtime(true) - APIX_START_TIME, 3) . ' seconds';
        }

        if (null !== $this->options['extras']) {
            $data['extras'] = $this->options['extras'];
        }

        $name = $this->options['name'];
        $response->results[$name] = $data;
    }
}
```

As you can see, this plugin will be activated at the beginning (`early`) of the `response` event and makes some edits to the response object.

Hooks

There are several events that a plugin may hook into. In order of when they fire, they are:

- server, early
- entity, early
- entity, late
- server, exception // if there is an exception
- response, early
- response, late

- server, late

Server Hook The Server hook allows a user to create plugins that run before and after an entity is run and the response objects are generated. The main server (`ApixServer`) is passed in to the plugin's update function.

Entity Hook The Entity Hooks fire before and after the required resource is called. For example, if you're using the class method for controllers and are calling `onRead()`, the entity hooks will fire immediately preceding and after that call. Plugins that use the entity hooks will receive the entity object as the parameter in their update function.

An example that uses the "entity, early" hook is the Authentication plugin which checks to see whether the requested resource is protected and then serves based on satisfying a permissions check.

Response Hook The Response hook is used to manipulate a response object following the completion of the entity calls. The "early" hook allows access to the response object before it is encoded into the requested format. The "late" hook allows access to the response object *after* is has been encoded. Plugins that use the response hooks will receive the response object as the parameter in their update function.

Some examples of plugins that use the "response, early" hook include the OutputDebug and OutputSign plugins. The Tidy plugin makes use of the "response, late" hook in order to clean up the response output after it has been encoded appropriately (into JSON, XML, HTML, etc).

Entity Plugins

Entity Plugins have the unique ability to access the method or closure annotations of the entities that they associate with. The annotations are parsed and then available for use in the `update` method. The annotation tag is defined using the `PluginAbtractEntity::$annotation` property of your plugin. APIx will then look in your entity definitions for the specified tag and parse out key=value pairs.

In the following example we'll write a very quick (and incomplete) plugin that logs usage if the entity is successfully called. The adapter should implement `My\Usage\LogAdapter` which in this example would have a `log` method which would, presumably, log usage. This plugin will use the @api_logusage annotation. If the annotation doesn't exist in the entity, this plugin will not call the adapter's log method.

```
<?php

namespace Apix\Plugin;

class UsageLogPlugin extends PluginAbstractEntity {

    public static $hook = array('entity', 'late');
    protected $options = array('adapter' => 'My\Usage\LogAdapter');
    protected $annotation = 'api_logusage'

    public function update(\SplSubject $entity) {
        $method = $this->getSubTagValues('method');
        $value = $this->getSubTagValues('value');

        if($method != null) {
            $this->adapter->log($method, $value);
        }
    }

}
```

An example entity that makes use of the above plugin might look like this:

```
<?php

use Apix\Request;

class Echo {

    /**
     * Echo out the data that was POSTed
     *
     * @return array
     * @api_logusage method=echo value=1
     */
    public function onCreate(Request $request) {
        $data = $request->getBodyData();
        return array("echo" => $data);
    }

}
```

Indices and tables

- *genindex*
- *search*

CHAPTER 2

Contributing

Contributing

Contributing to the documentation

Author My Name

You can edit the docs online by clicking the "**edit this page**" button on any given page which will open an online editor for that page.

Or you can fork the Github's repo, add your modifications, followed by a pull request.

Formatting guidelines

The documentations are written and comply to the reStructuredText format which is a plain text markup syntax similar to Markdown. See the official Quick Reference.

Basics

First, always use UTF-8 (with BOM) as encoding and 4-spaces (no tabs) for indentation.

Text should be wrapped at 80 columns. The only exception should be long URLs, and code snippets.

Headings

Heading are are created by underlining the title with the following characters:

- = Headline 1 for the page title.
- – Headline 2 for the page sections.
- ^ Headline 3 for sub-sections.
- ~ Headline 4 for sub-sub-sections.
- " Headline 5 for sub-sub-sub-subsections

The length of the underline must be as long as the headline.

The headings are preceded and followed by a blank line.

Paragraphs

Paragraphs are simple blocks of text, with all the lines at the same level of indentation. Paragraphs should be separated by more than one empty line.

Inline styles

- `*italics*` for *italics*.
- `**bold**` for **bold**.
- ``` ``code`` ``` for `code or fixed-space literals`.

If you need to escape a character use a backslash e.g. *.

Lists

List markup is very similar to Markdown.

Unordered lists

```
* This is a bullet.
    * This is a sub bullet.
```

Numbered lists

```
1. First line
2. Second line
#. Use ``#`` for auto numbering.
#. Would be listed as 4.
```

Description lists (aka <dl>)

```
term/name (aka <dt>)
    definition (aka <dd>)
```

Links

External links

`` `Info.com <http://www.info.com>`_ `` produces an external link to Info.com.

Internal pages

`:doc:`

`:doc:`documentation`` produces `documentation`.

Also, works with absolute path. Omit the `.rst` extension.

Cross referencing
:ref:

For instance, using `:ref:`some-ref-label`` produces *Cross referencing*.

Fro this to work, it needs a *unique* reference label target such as:

```
.. _some-ref-label:

Cross referencing
~~~~~~~~~~~~~~~~~
```

Note the reference label must be unique across the entire doc. Using `dir-page-section` and `namespace-class-method` is a good practice.

Adding image/figure files

```
.. figure:: ../images/picture.gif
    :scale: 50 %
    :alt: Some alternative text.
```

Check the relevant directive for more info.

Stylesheet

Individual CSS styles will eventually be referenced here.

Highlighted block

Special notices/blocks of information can highlighted such as:

Tips

Use `.. tip::` to document or re-iterate interesting points.

Tip: This is a helpful tid-bit you probably forgot.

Notes

Use `.. note::` to document some important piece of information.

Note: This is a special note.

Warnings

Use `.. warning::` to document potential stumbling blocks.

Warning: Ths is a warning – potential hazard!

Describing language specific features

Use the following for language specific directives:

PHP

The PHP related documentation use the phpdomain to provide custom directives for describing PHP objects and constructs.

Describing classes and constructs Each directive populates the index, and or the namespace index.

`.. php:global::` name
: This directive declares a new PHP global variable.

`.. php:function::` name(signature)
: Defines a new global function outside of a class.

`.. php:const::` name
: This directive declares a new PHP constant, you can also use it nested inside a class directive to create class constants.

`.. php:exception::` name
: This directive declares a new Exception in the current namespace. The signature can include constructor arguments.

`.. php:class::` name
: Describes a class. Methods, attributes, and constants belonging to the class should be inside this directive's body:

```
.. php:class:: MyClass

    Class description

    .. php:method:: method($argument)

    Method description
```

Attributes, methods and constants don't need to be nested. They can also just follow the class declaration:

```
.. php:class:: MyClass

    Text about the class

.. php:method:: methodName()

    Text about the method
```

See also:

php:method, php:attr, php:const

`.. php:method::` name(signature)
: Describe a class method, its arguments, return value, and exceptions:

```
.. php:method:: instanceMethod($one, $two)

    :param string $one: The first parameter.
    :param string $two: The second parameter.
    :returns: An array of stuff.
```

```
        :throws: InvalidArgumentException

       This is an instance method.
```

.. **php:staticmethod::** `ClassName::methodName(signature)`
Describe a static method, its arguments, return value and exceptions, see `php:method` for options.

.. **php:attr::** `name`
Describe an property/attribute on a class.

CHAPTER 3

Internal API Doc

This is the extracted internal API for APIX.

Apix

Contents:

Apix\Main

class Apix**Main**

Main:: VERSION = '@package_version@';

property $config

Todo review $his.

Var array

property $request

Var Request

property $route

Var Route

property $resources

Var Resources

property $entity

Var Entity

property $response

Var Response

__construct()
Constructor.

Returns void

run()
Run the show...

Throws \InvalidArgumentException $04

CodeCoverageIgnore

getServerVersion()
Gets the server version string.

Returns string

setRouting()
Sets and initialise the routing processes.

Parameters

- **$request** (*Request*) –

Returns void

getRoute()
Returns the route object.

Returns Router

getResponse()
Returns the response object.

Returns Response

negotiateFormat()
Returns the output format from the request chain.

•[default] => string e.g. 'json',

•[controller_ext] => boolean,

•[override] => false or $_REQUEST['format'],

•[http_accept] => boolean.

Parameters

- **$opts** (*array*) – Options are:
- **$ext** (*string|false*) – The contoller defined extension.

Returns string

Apix\Plugin\PluginAbstract

class Apix\Plugin**PluginAbstract**

property $adapter
Holds a plugin's adapter.

Var closure|object

property $options
Holds an array of plugin's options.

Var array

__construct()
Constructor

Parameters

- **$options** (*mix*) – Array of options

checkAdapterClass()
Checks the plugin's adapter comply to a class/interface

Parameters

- **$adapter** (*object*) –
- **$class** (*object*) –

Throws \RuntimeException

Returns true

setOptions()
Sets and merge the defaults options for this plugin

Parameters

- **$options** (*mix*) – Array of options if it is an object set as an adapter

getOptions()
Gets this plugin's options

Returns array

setAdapter()
Sets this plugin's adapter

Parameters

- **$adapter** (*closure|object*) –

getAdapter()
Gets this plugin's adapter

Returns mix

log()
Just a shortcut for now. This is TEMP and will be moved elsewhere! TODO: TEMP to refactor

Apix\Server

class Apix**Server**

onCreate()
POST request handler

Parameters

- **$path** (*string*) – The path name to match against.
- **$to** (*mixed*) – Callback that returns the response when matched.

See Server::proxy

Returns Controller $rovides a fluent interface.

onRead()
GET request handler

Parameters

- **$path** (*string*) – The path name to match against.
- **$to** (*mixed*) – Callback that returns the response when matched.

See Server::proxy

Returns Controller $rovides a fluent interface.

onUpdate()
PUT request handler

Parameters

- **$path** (*string*) – The path name to match against.
- **$to** (*mixed*) – Callback that returns the response when matched.

See Server::proxy

Returns Controller $rovides a fluent interface.

onModify()
PATCH request handler

Parameters

- **$path** (*string*) – The path name to match against.
- **$to** (*mixed*) – Callback that returns the response when matched.

See Server::proxy

Returns Controller $rovides a fluent interface.

onDelete()
DELETE request handler

Parameters

- **$path** (*string*) – The path name to match against.
- **$to** (*mixed*) – Callback that returns the response when matched.

See Server::proxy

Returns Controller $rovides a fluent interface.

onHelp()
OPTIONS request handler

Parameters

- **$path** (*string*) – The path name to match against.
- **$to** (*mixed*) – Callback that returns the response when matched.

See Server::proxy

Returns Controller $rovides a fluent interface.

onTest()
HEAD request handler

Parameters

- **$path** (*string*) – The path name to match against.
- **$to** (*mixed*) – Callback that returns the response when matched.

See Server::proxy

Returns Controller $rovides a fluent interface.

proxy()

Acts as a shortcut to resources::add.

when matched.

See Resources::add

Parameters

- **$method** (*string*) – The HTTP method to match against.
- **$path** (*string*) – The path name to match against.
- **$to** (*mixed*) – Callback that returns the response

Returns Controller

setGroup()

Test Read from a group (TODO).

Parameters

- **$opts** (*array*) – Options are:

Returns string

a

Symbols

A

C

D

E

G

L

M

N

O

P

R

S