Comodojo dispatcher Documentation

Release 3.0.0

Marco Giovinazzi

December 15, 2015

Comodojo dispatcher is a service-oriented REST microframework designed to be simple to use, extensible and fast.

Before version 3 project name was "SimpleDataRestDispatcher" and it was mainly oriented to JSON and XML pure-data REST services. Since version 3.0, dispatcher has become a totally different project, designed to publish and manage (almost) any kind of data.

Dispatcher is structured on top of following main concepts:

- A dispatcher instance is a sort of multi-service container; services are grouped in bundles, managed by composer and installed from the project package. Installing or removing a bundle should never stop or interfere with other bundles or the dispatcher itself.
- Services are the central point of framework's logic: they are independent, callable php classes that may return data (in any form) or errors (exception); a service must extend the *ComodojoDispatcherService* class.
- Routes are paths (urls) that may be associated to a service, redirect to another location or generate errors (exception); in practice, routes are only paths that forward a request to a service without any knowledge of the service's logic.
- Services and routes are completely separated. It means that a single service may be reached via multiple routes and it's life does not depend on any of them.
- In dispatcher, (almost) everything about requests, routes and results can be modified using events' subsystem; plugins are made to hook those events and interact with the framework. They can also be packed in bundles and managed using composer.

Contents:

CHAPTER 1

Installation

Comodojo dispatcher could be installed via composer, using dedicated dispatcher.project package.

1.1 Requirements

To work properly, dispatcher requires an apache webserver with PHP >=5.3.0, installed as apache module or cgi/fastcgi.

It may work on different webservers like nginx (ensure to convert the .htaccess logic if you plan to use rewrite mode), but this is actually untested.

1.2 Installing via composer

First install composer, then create a new dispatcher.project using this command:

```
php composer.phar create-project comodojo/dispatcher.project dispatcher
```

This will install a new instance of dispatcher and required dependencies in “dispatcher” folder.

If you need also default content and tests, install the package:

```
php composer.phar require comodojo/dispatcher.servicebundle.default
```

## Downloading as archive

Stable releases are published on dispatcher.comodojo.org.

To install, download latest package and type (in the package folder):

```
php composer.phar install
```

CHAPTER 2

Configuration

This section covers the framework's configuration if installed via dispatcher.project package.

If you are using dispatcher.framework as a library, you should define constants, directives and folders manually.

2.1 Directory structure

The project package will create the following directory structure (excluding files):

```
dispatcher/
    - DispatcherInstaller/ => contains the DispatcherInstaller class, fired when composer install
    - cache/ => cache files will be written here; ensure apache user can write here
    - configs/ => configuration files' folder
    - logs/ => where monolog (if enabled) will write logs
    - plugins/ => this folder is dedicated to plugins manually installed or created by user
    - services/ => this folder is dedicated to services manually installed or created by user
    - templates/ => this folder is dedicated to templates manually installed or created by user
    - vendor/ => composer standard vendor folder
    - index.php
    - [...]
```

Note: It's important to understand that plugin/service/template packages will be installed in *vendor* folder, respecting the composer installation standard. This because mixing user files and package files could be a not optimal solution to handle updates or customization. The framework can address services and plugins using a relative/absolute path convention, to keep installer aware of where packages are located.

2.2 Configuration files

Dispatcher comes out of the box with a default configuration that can be edited to change global behaviour of framework.

Configuration files are contained in *configs* folder:

- *dispatcher-config.php*: contains constants defined to change some aspect of framework (like paths, logging, ...)
- *plugins-config.php*: (should) contain plugins init scripts; initially empty
- *routing-config.php*: (should) contain declaration of routes; initially empty

These files are loaded at boot time.

2.3 General properties

2.3.1 DISPATCHER_REAL_PATH

Dispatcher real path.

```
define("DISPATCHER_REAL_PATH",realpath(dirname(__FILE__))."/../");
```

2.3.2 DISPATCHER_BASEURL

Dispatcher baseurl. If not defined, dispatcher will try to resolve absolute base url itself (default).

```
define("DISPATCHER_BASEURL","");
```

2.3.3 DISPATCHER_ENABLED

If false, dispatcher will not route any request and will reply with an *503 Service Temporarily Unavailable* status code.

```
define ('DISPATCHER_ENABLED', true);
```

2.3.4 DISPATCHER_USE_REWRITE

If true, dispatcher will use rewrite module to acquire service path and attibutes.

If you prefer to turn this feature off, remember to remove/rename .htaccess file in installation's folder and/or disable the apache rewrite module.

```
define ('DISPATCHER_USE_REWRITE', true);
```

2.3.5 DISPATCHER_AUTO_ROUTE

Enable/disable the autoroute function; if true, dispatcher will try to route requests to not declared services using filenames.

```
define('DISPATCHER_AUTO_ROUTE', false);
```

2.3.6 DISPATCHER_DEFAULT_ENCODING

Sets the system-wide default encoding.

```
define('DISPATCHER_DEFAULT_ENCODING', 'UTF-8');
```

2.3.7 DISPATCHER_SUPPORTED_METHODS

HTTP supported methods.

This represent the pool of framework-supported HTTP methods, but each service can implement one or more methods independently. This value may change the *Allow Response* Header in case of 405 response.

Change this value only if: - you need to support other http methods (like PUSH) - you want to disable globally a subset of HTTP methods (i.e. if you want to disable PUT requests globally, you can omit it from this definition; method will be ignored even though service implements it - or implements the *ANY* wildcard).

Note: a service that not implements one of this methods, in case of unsupported method request, will reply with a *501-not-implemented* response; this behaviour is managed automatically.

Warning: this constant should be in plain, uppercased, comma separated, not spaced text.

Warning: DO NOT USE a "ANY" method here or it will override the embedded wildcard ANY.

```
define('DISPATCHER_SUPPORTED_METHODS', 'GET,PUT,POST,DELETE');
```

2.4 Logging

2.4.1 DISPATCHER_LOG_ENABLED

enable/disable logger (monolog).

```
define('DISPATCHER_LOG_ENABLED', false);
```

2.4.2 DISPATCHER_LOG_NAME

Log channel name.

```
define('DISPATCHER_LOG_NAME', 'dispatcher');
```

2.4.3 DISPATCHER_LOG_TARGET

Log target (file or *null* for error_log).

```
define('DISPATCHER_LOG_TARGET', null)
```

2.4.4 DISPATCHER_LOG_LEVEL

Debug level, as in psr-3 standard.

```
define('DISPATCHER_LOG_LEVEL', 'ERROR')
```

2.5 Folders

2.5.1 DISPATCHER_CACHE_FOLDER

Cache folder.

```
define('DISPATCHER_CACHE_FOLDER', DISPATCHER_REAL_PATH."cache/");
```

2.5.2 DISPATCHER_SERVICES_FOLDER

Services folder.

```
define('DISPATCHER_SERVICES_FOLDER', DISPATCHER_REAL_PATH."services/");
```

2.5.3 DISPATCHER_PLUGINS_FOLDER

Plugins folder.

```
define('DISPATCHER_PLUGINS_FOLDER', DISPATCHER_REAL_PATH."plugins/");
```

2.5.4 DISPATCHER_TEMPLATES_FOLDER

Templates folder.

```
define('DISPATCHER_TEMPLATES_FOLDER', DISPATCHER_REAL_PATH."templates/");
```

2.5.5 DISPATCHER_LOG_FOLDER

Logs folder.

```
define('DISPATCHER_LOG_FOLDER', DISPATCHER_REAL_PATH."logs/");
```

2.6 Cache

2.6.1 DISPATCHER_CACHE_ENABLED

Enable/disable cache support.

```
define('DISPATCHER_CACHE_ENABLED', true);
```

2.6.2 DISPATCHER_CACHE_DEFAULT_TTL

Default cache time to live, in seconds.

```
define('DISPATCHER_CACHE_DEFAULT_TTL', 600);
```

2.6.3 DISPATCHER_CACHE_FAIL_SILENTLY

If true, cache will fail silently in case of error without throwing exception.

```
define('DISPATCHER_CACHE_FAIL_SILENTLY', true);
```

CHAPTER 3

How dispatcher works

(page yet to be written)

CHAPTER 4

Writing services

(page yet to be written)

CHAPTER 5

Routing requests

Dispatcher includes a minimal embedded URL router that maps urls to services, using an approach slightly different from the one used in other common framework.

Instead of map methods (or functions) to full uris (plus HTTP verbs), dispatcher retrieves routes using only the first path in the URI. In practice, this variable *select* the route the request will follow.

This kind of next-hop approach leaves all logic to the service itself (for example, the ability to serve POST requests instead of GET), decoupling routing and service processing.

Note: To understand how to handle HTTP methods jump to the service section.

5.1 mod_rewrite

Dispatcher uses the apache mod_rewrite extension to acquire requests and route those to relative services.

It is also possible to disable this feature and force dispatcher to reply only to requests directed to index.php, settting the *DISPATCHER_USE_REWRITE* constant to false.

In this case, parameters should be passed as a standard http query-string *key=value” pairs.

5.2 Attributes and Parameters

TBW

5.3 Defining routes

To define a new route, the *setRoute()* method should be invoked before the *dispatch()*.

Syntax of method is:

```
setRoute( [route], [type], [target], [parameters], [relative] )
```

So, an example route could be:

```
$dispatcher->setRoute("helloworld", "ROUTE", "HelloWorld.php", array(), true);
```

5.4 Predefined routes

The router supports 2 special routes:

- Landing route “” (empty string)
- Default route “default”

Only the default route is initially defined and lands to a 404 “Service not found” error.

5.5 Autorouting

If enabled setting constant *DISPATCHER_AUTO_ROUTE*, dispatcher will try to map requests to service files using file names.

Only files in the *DISPATCHER_SERVICES_FOLDER* are taken into account.

5.6 Conditional routing

Thanks to the event subsystem, dispatcher can force or totally override the routing logic.

This snippet (from dispatcher.plugin.test) simply change the target service if a special request header is provided.:

```
public static function conditional_routing_header($ObjectRoute) {

    $headers = self::getRequestHeaders();

    if ( array_key_exists("C-Conditional-Route", $headers) ) {

        $ObjectRoute
            ->setClass("test_route_second")
            ->setTarget("vendor/comodojo/dispatcher.servicebundle.test/services/test_route_second

    }

    return $ObjectRoute;

}
```

5.7 Router-side attributes inject

TBW

CHAPTER 6

The event system

Dispatcher has an integrated event's system that can be used to extend its features by plugins.

When a client calls dispatcher, request, route and response are modeled as objects and provided to the callback function hooked to relative event.

Dispatcher starts to emit events as soon as a request is received.

Let's consider an example:

```
global $dispatcher;

function custom_404($ObjectError) {

    $error_page = file_get_contents(DISPATCHER_REAL_PATH."vendor/comodojo/dispatcher.plugin.test/

    $ObjectError->setContent($error_page);

    return $ObjectError;

}

$dispatcher->addHook("dispatcher.error.404", "custom_404");
```

In this example, a plugin will set a custom content in default 404 error page by:

- catching event "dispatcher.error.404";
- replacing error content using "setContent()" method;
- returning to framework the modified object.

6.1 How dispatcher emits events

By default, dispatcher uses this schema to emit events:

[*framework*].[*event*].[*notification**|*marker*]

In practice:

- [framework] will always be populated by the string *dispatcher*;
- [event] represents the event macroclass; possible values are *request*, *routingtable*, *serviceroute*, *result*, *route*, *redirect* or *error*.
- [notification|marker] is a detailed view of what is happening; it can assume different values, like HTTP return codes (like **404** in the previous example for a "not found" response.

There are also two special category of events:

- start event (*dispatcher*), that will fire at framework startup;

- markers, fired to express a particular condition (like # that denote the end of specific event's macroclass).

Each type of event expects the callback to return a particular kind of object. If something different is provided, callback's result will be discarded.

Note: If a single event is hooked to multiple callbacks, it will behave as a chain: the first result (if any) will be the input of the second callback and so on.

Sone examples are:

- *dispatcher.serviceroute* - a level 2 event that expose the route retrieved for the current request
- *dispatcher.error.404* - a level 3 event for a not found response
- *dispatcher.result.#* - a level 3 event that fires after every other callback

6.2 The complete event list

- dispatcher receive a request
  - *dispatcher* - marks the frameworks has entered the running cycle and exposes the whole *ComodojoDispatcherDispatcher* instance without expecting any return value
- Request is modeled as an instance of *ComodojoDispatcherObjectRequestObjectRequest*
  - *dispatcher.request* - provides and expects an instance of *ComodojoDispatcherObjectRequestObjectRequest*
  - *dispatcher.request.[METHOD]* - provides and expects an instance of *ComodojoDispatcherObjectRequestObjectRequest*
  - *dispatcher.request.[SERVICE]* - provides and expects an instance of *ComodojoDispatcherObjectRequestObjectRequest*
  - *dispatcher.request.#* - provides a *ComodojoDispatcherObjectRequestObjectRequest*, will fire after every other callback discarding returned data
- An instance of *ComodojoDispatcherObjectRoutingTableObjectRoutingTable* is created
  - *dispatcher.routingtable* - provides and expects an instance of *ComodojoDispatcherObjectRoutingTableObjectRoutingTable*
- A route was retrieved from routingtable
  - *dispatcher.serviceroute* - provides and expects an instance of *ComodojoDispatcherObjectRouteObjectRoute*
  - *dispatcher.serviceroute.[TYPE]* - provides and expects an instance of *ComodojoDispatcherObjectRouteObjectRoute*
  - *dispatcher.serviceroute.[SERVICE]* - provides and expects an instance of *ComodojoDispatcherObjectRouteObjectRoute*
  - *dispatcher.serviceroute.#* - provides a *ComodojoDispatcherObjectRouteObjectRoute*, will fire after every other callback and discarding returned data
- Once a route is retrieved, dispatcher will run a service, redirect to another location or return an error; in any case, a result object that implements the *ComodojoDispatcherObjectResultObjectResultInterface* is inited, provided to the callback and expected as result.
  - *dispatcher.result*
  - In case of success
    - *dispatcher.route*
    - *dispatcher.route.[STATUSCODE]*

  - In case of redirect
    * *dispatcher.redirect*
    * *dispatcher.redirect.[STATUSCODE]*
  - In case of error
    * *dispatcher.error*
    * *dispatcher.error.[STATUSCODE]*
  - *dispatcher.result.#* will fire after every other callback and expects an instance of *ComodojoDispatcher-ObjectResultObjectResultInterface*
- result is returned to client

CHAPTER 7

Plugins

(page yet to be written)