Comodojo dispatcher Documentation

Release 4.0.0

Marco Giovinazzi

Feb 10, 2019

Contents

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

Table of Contents:

CHAPTER 1

General Concepts

The dispatcher framework is a library designed to be extensible and hackable enough to support the development of web resources, from plain REST services/APIs to web application.

Unless other complex and wider frameworks, it offers only the minimal set of tools to process and respond to web requests, leaving to the developer the choice to use one programming pattern or another, one library or another equivalent one.

This minimal set of features currently includes:

- web calls modelling (from request to response);
- service models;
- application routing (using extended regular expressions);
- events management and delegation;
- HTTP post processing (content, headers and status code);
- resource caching;
- auto packaging (bundles) and configuration.

1.1 Web calls modelling

Once started (i.e. when a new HTTP request hits the web server that sends is to the dispatcher engine), the dispatcher starts immediately creating models for the request, the routing engine and the response, plus a couple of classes for configuration, extra parameters, cache and events. In other words, the entire workflow from a request to the response is mapped to the internal dispatcher functional schema, accessible using internal APIs.

In the middle of this workflow there is the service with its business logic, that has to run to provide value to the client.

The service and the event listeners are the only entities that can interact with the models' APIs to create the output.

1.2 Service models

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).

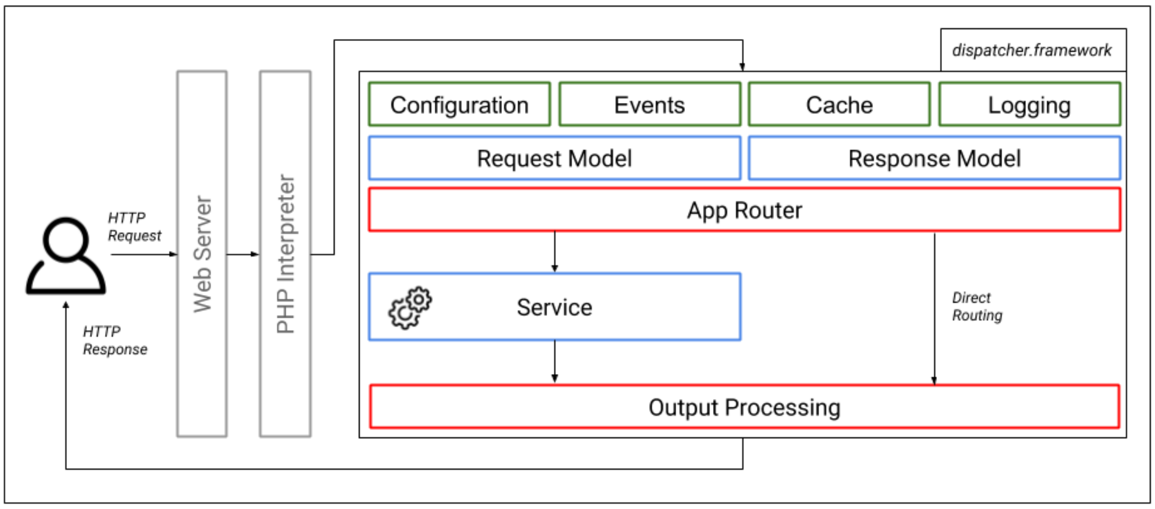

Fig. 1: comodojo/dispatcher-framework v4.X architecture

Note: A dispatcher instance is a sort of multi-service container; services can be grouped in bundles (packages) with auto-describing metadata and installed automatically. This way installing or removing a bundle should never stop or interfere with other bundles or the dispatcher itself.

Services a created on top of the service model that:

- defines basic components for the service (e.g. http verbs support);
- provides complete access to all the framework's functionalities.

A service is selected evaluating a route definition in the routing table (see next paragraph). However, *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.

For more information about services, jump to the *Writing services* section.

1.3 Application routing

The dispatcher framework includes an advanced URL router that maps urls to services and allows users to describe routes using regular expressions, evaluate and convert url paths into parameters.

Since the services and the routes are loosely coupled, the dispatcher's app router acts more or less like an IP router: it owns a routing table that is queried to determine where to forward the execution flows (which class is responsible for what route) and expects a return value from it, but nothing more.

Also, the configuration of the routing table is not included in the service classes, but in a specific configuration file that is loaded and cached at boot time.

For more information about the dispatcher app router, jump to the *Routing requests* section.

Additionally, to support automatic configuration and packaging, definition of routes can be included in the bundle metadata and processed by the comodojo/installer package (see *Auto packaging and auto configuration*).

1.4 Events management

Dispatcher is an event-based framework: at every step of its lifecycle dispatcher is emitting events, and event listeners (plugins) can be used to hook to the events, access the framework APIs and change its behaviour. This means that a plugin can change almost everything in the framework, even the request object.

Let's consider a couple of common use cases.

Authenticating requests

In a typical scenario, a subset of the exposed servics requires the client to be authenticated using a JWT token to consume the data.

Dispatcher does not include authentication out of the box, but a custom one (or perhaps an external library) can be integrated using events. To perform the authentication without changing the service specs, a custom listener could:

- catch the *dispatcher.route* event;
- determine the service route;
- check if the service requires authentication;
- perform authentication and, in case, raise a 401 Unauthorized HTTP error.

Change output encoding

If one of your client does not support the default output encoding of a service, you can write a plugin to:

- catch the *dispatcher.response* event;
- check if the output should be changed according to the client's IP or name;
- in case, change the output encoding.

Also in this case there is no need to change the service specs, but just hack the dispatcher to behave differently for the particular source.

As for the services, plugins can be packed in bundles, reused and auto-installed by the comodojo/installer package.

For more information about events, jump to the events section.

1.5 HTTP post processing

TBW

1.6 Resource caching

Dispatcher is shipped with the comodojo/cache package that is responsible to:

- cache internal structures (e.g. the routing table);
- cache (if required) output data (server side);
- cache user-defined data.

About output data, dispatcher will check if (i) the service is configured support cache and (ii) the HTTP verb allows caching. In this case, the complete result object will be stored in the cache provider and used to build the response to the next similar requests.

For more information about resource caching, jump to the *Enabling cache* section.

1.7 Auto packaging and auto configuration

TBW

CHAPTER 2

Installation

The comodojo dispatcher framework can be installed using composer as a product (using the dedicated dispatcher project package) or as a library.

2.1 Install dispatcher as a product

To install dispatcher as a product, simply run the composer create-project command (assuming *dispatcher* as your project folder):

```
composer create-project comodojo/dispatcher dispatcher
```

Composer will install the dependencies and create the main configuration file. Once completed, configure the web server to use *dispatcher/public* as the document root.

2.2 Install dispatcher as a library

To install dispatcher as a library in your own project:

```
composer require comodojo/dispatcher.framework
```

Note: If installed as a standalone library, no automatic configuration will be performed by composer. To create a custom project based on dispatcher is highly recommended to start cloning and changing the dispatcher project package.

2.3 Requirements

To work properly, dispatcher requires a web server and PHP >= 5.6.0.

CHAPTER 3

Basic usage

The easiest way to use the dispatcher framework is installing it from the comodojo/dispatcher project package. In this case, the framework is almost ready to use: services and plugins have dedicated folders (pre-configured for autoloading), the main configuration file is created automatically and rewrite rules are in place (only for apache web server).

Alternatively, dispatcher can be integrated in a custom project simply installing it as a library. In this case, there are few steps to follow in order to start using it (see).

3.1 The dispatcher project package

Note: This section assumes that you have installed dispatcher using the dispatcher project package. If you are using dispatcher as a library in your custom project, skip this section and continue with *Using dispatcher in custom projects*.

The comodojo/dispatcher is the default project package for the dispatcher framework. It includes a standard folder structure, a default set of CLI command and the comodojo/comodojo-installer package.

Once installed, it creates the following folder structure:

::

[base folder]/ /cache /commands /config /logs /plugins /public /services

The document root

The **public** folder is the document root directory. It contains the *index.php* file to start the framework and the *.htaccess* file to configure the rewrite rules in Apache server.

To start dispatcher, **configure the document root of your server to this location**.

Configuration files

Configuration files are in the **config** directory, in YAML format:

- *comodojo-configuration.yml*, that contains the global configuration (e.g. cache, logs, . . .)
- *comodojo-routes.yml*, the initial configuration of the routing table
- *comodojo-plugins.yml*, plugins configuration
- *comodojo-commands.yml*, commands configuration

More information on the configuration files in the :ref:config section.

Services, plugins and commands

The three folders **services**, **plugins** and **commands** are there to host your custom services, plugins and commands, respectively. Since these folders are included in the autoloader, the custom classes have to be implemented using the declared class path: `\Comodojo\Dispatcher\Services\` for services, `\Comodojo\Dispatcher\Plugins\` for plugins and `\Comodojo\Dispatcher\Command\` for commands.

Temp files

THe **cache** and the **log** folder are there only to host temporary files for cache (i.e. in case of a file-based cache provider) and logs (i.e. in case of logging enabled). In production installation, however, it's suggested to configure the framework to put those files in the OS default directories (e.g. */tmp* for cache and */var/log* for log files).

3.2 Your first hello world application

To create your first hello world application using dispatcher, there are essentially two steps to follow:

1. create the hello world service;
2. configure the route to the service.

3.2.1 HelloWorld service

An example HelloWorld service that implements (only) the HTTP GET method can be implemented as:

```php
<?php namespace Comodojo\Dispatcher\Services;

use \Comodojo\Dispatcher\Service\AbstractService;

class HelloWorld extends AbstractService {

    // HelloWorld is available ONLY via HTTP GET
    // Other HTTP verbs are not supported and, if requested, the framework
    // returns a "501 Method Not Implemented" response
    public function get() {

        // Access the request and the query component
        $query = $this->getRequest()->getQuery();

        // Get the "to" attribute
        $to = $query->get('to');

        // return content based on the received attribute
        return empty($to) ? 'Hello Comodojo!' : "Hello $to!";
```

(continues on next page)

(continued from previous page)

```

    }

}
```

Save this file as *HelloWorld.php* in the *services* folder. Since this path is registered in the autoloader for the `\Comodojo\Dispatcher\Services` namespace, the class will become immediately auto-loadable.

For more information about services, jump to the *Writing services* section.

3.2.2 Configure the route to the service

Now that a service is available, we have to install a new route in the dispatcher router. The service accepts only one optional parameter **to** and we can use a regex to validate this parameter. For example, we can allow only alphanumeric chars, underscore and spaces, in any combination. Our route will be:

Base Path	Variable Path
helloworld/	{"to":"^[**a-zA-Z0-9_**\s]+$"}

This route can be installed adding one entry in the *comodojo-routes.yml* file:

```
helloworld:
    type: ROUTE
    class: \Comodojo\Dispatcher\Services\HelloWorld
    parameters: {}
    route: 'helloworld/{"to":"^[a-zA-Z0-9_\\s]+$"}'
```

If the file does not exist, create it as *config/comodojo-routes.yml*.

For more information about routes, jump to the *Routing requests* section.

3.3 Adding a plugin

Now that our HelloWorld service is in place, we can add a plugin to modify the global behaviour of the framework if one or more conditions are met.

An example could be: if (i) a route is matched, (ii) the route leads to the **HelloWorld** service and (iii) the request contains **text/html** in the *Accept* header then (iv) change the content-type to **text/html** and (v) wrap the text into a *<h1>* tag.

The plugin code will be:

```
<?php namespace Comodojo\Dispatcher\Plugins;

use \League\Event\AbstractListener;
use \League\Event\EventInterface;

class HelloWorldPlugin extends AbstractListener {

    public function handle(EventInterface $event) {

        $request = $event->getRequest();
        $response = $event->getResponse();
        $route = $event->getRouter()->getRoute();

        if (
            // (i) a route is matched
```

(continues on next page)

(continued from previous page)

```
                $route !== null &&
                // (ii) the route leads to the **HelloWorld** service
                $route->getClassName() === '\Comodojo\Dispatcher\Services\HelloWorld' &
↪&
                // (iii) the request contains **text/html** in the *Accept* header
                strpos($request->getHeaders()->get('Accept'), 'text/html') !== false
            ) {
                $content = $response->getContent()->get();
                // (iv) change the content-type to **text/html**
                $response->getContent()->setType('text/html');
                // (v) wrap the text into a <h1> tag
                $response->getContent()->set("<h1>$content</h1>");
            }

        }

}
```

For more information about plugins and events, jump to the *Plugins* section.

3.4 Using dispatcher in custom projects

When used as a library in a custom project, dispatcher cannot rely on the pre-defined loader and also on the automation offered by the comodojo/comodojo-installer package.

For the above mentioned reasons, there are some steps that are needed to make the framework work:

Define your own folder structure

The dispatcher framework does not require any specific folder: dispatcher will work even if all the code is stored in a flat folder.

However, a good practice could be to start cloning the comodojo/dispatcher package and then add, change or delete files or folders according to your needs.

Of course, another good practice is to create a document folder that is isolated from the code, but this is still up to you.

Note: The only limitation is the name of the loader file (see next sections), that is autoprocessed by the framework to understand the actual absolute URI.

Add dispatcher as a dependency in your *composer.json*

To add the dispatcher framework as a dependency, follow the *Install dispatcher as a library* section.

Writing the loader (*index.php* file)

The *index.php* is the actual entry point to the framework, and its name is is the only limitation imposed by dispatcher.

This file is in charge of:

- creating an instance of dispatcher;

- adding plugins and routes;
- triggering the `Dispatcher::dispatch()` method.

Creating a new instance of dispatcher is trivial:

```
use \Comodojo\Dispatcher\Dispatcher;
$dispatcher = new \Comodojo\Dispatcher\Dispatcher([
    // configuration parameters here!
]);
```

The class constructor supports optional parameter for: - events manager (An instance of the ComodojoFoundationEventsManager class); - cache manager/provider (PSR-16 compliant); - logger (PSR-1 compliant).

If no optional parameter is specified, dispatcher will create default (empty) object for you.

Plugin can be installed using the EventManager:

```
$manager =  $dispatcher->getEvents();
```

Routes have to be pushed in the routing table:

```
$table =  $dispatcher->getRouter()->getTable();
```

If no routes are provided, by default dispatcher will reply a *404 - Not found* error to all requests.

Note: For more information about routes, see the *Routing requests* section.

Rewrite rules

Dispatcher relies on rewrite rules to work correctly. The rewrite routes shall be placed in the document folder at the same level of the *index.php* loader.

An example rewrite rule (the one that the comodojo/dispatcher package uses by default) is the following for apache:

```
<IfModule mod_rewrite.c>

    <IfModule mod_negotiation.c>
        Options -MultiViews
    </IfModule>

    Options +FollowSymLinks
    IndexIgnore */*

    RewriteEngine On

    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d

    RewriteRule (.*) index.php [L]
</IfModule>
```

Or the equivalent version for nginx could be:

```
location / {
  if (!-e $request_filename){
    rewrite ^(.*)$ /index.php break;
  }
}
```

CHAPTER 4

Configuration

The dispatcher framework can be configured passing configuration statements as an associative array when creating a new `\Comodojo\Dispatcher\Dispatcher` instance.

```
use \Comodojo\Dispatcher\Dispatcher;
$dispatcher = new \Comodojo\Dispatcher\Dispatcher([
    // configuration parameters here!
]);
```

A init time, the configuration is used to create basic objects (e.g. models, cache provider) and can be changed at any time accessing the configuration object (that will be an instance of ComodojoFoundationBasicConfiguration class).

```
// get the configuration object
$configuration = $dispatcher->getConfiguration();

// get the base-path value from actual configuration
$base_path = $configuration->get('base-path');

// set the my-param configuration item using standard notation
$configuration->set('my-param', 'foo');

// set the my->remote->url nested configuration item using dot notation
$configuration->set('my.remote.url', 'https://example.com')
```

Note: For more information about the `\Comodojo\Foundation\Base\Configuration` class, see the comodojo/foundation documentation

4.1 Configuration parameters

Following the list of configuration statement currently supported by the framework.

(bool) enabled [default: true]

Enable or disable the framework (i.e. if not enabled, dispatcher cannot serve any request).

(int) disabled-status [default: 503]

In case not enabled, the HTTP status to return to clients.

(string) disabled-message [default: Dispatcher offline]

In case not enabled, the HTTP body to return to clients.

(string) encoding [default: UTF-8]

Active char encoding.

(bool) routing-table-cache [default: true]

Enable or disable the caching of the routing table.

(int) routing-table-ttl [default: 86400]

The ttl for the routing table cache.

(array) supported-http-methods (default: null)

This setting could be used to restring (or enhance) the support for HTTP methods.

If not defined, the framework will allow the following http verbs:

- GET
- PUT
- POST
- DELETE
- OPTIONS
- HEAD
- TRACE
- CONNECT
- PURGE

(array) log

Logger configuration, input for the `LogManager::createFromConfiguration()` method (for more information, see the comodojo/foundation documentation)

An example schema:

```
log:
    enable: true
    name: dispatcher
    providers:
        local:
            type: StreamHandler
            level: debug
            stream: logs/dispatcher.log
```

(array) cache

Cache manager of provider configuration, input for the `SimpleCacheManager::createFromConfiguration()` method (for more information, see the comodojo/cache documentation)

An example schema:

```
cache:
    enable: true
    pick_mode: PICK_FIRST
    providers:
        local:
            type: Filesystem
            cache_folder: cache
```

4.2 Automatic configuration parameters

The following (basic) configuration parameters are computed and included in the configuration at init time.

Note: The user configuration has precedence over the automatic one: if one automatic parameter is included in the user configuration, its value will overwrite the automatic one.

(string) base-path

The base path of the project directory (i.e. the root of the *vendor* folder).

(string) base-url

The current URL used to contact the framework.

(string) base-uri

The full URI used to contact the framework.

(string) base-location

The relative path before the entry point (i.e. the *index.php* file).

CHAPTER 5

The request object

CHAPTER 6

Routing requests

The dispatcher framework includes an advanced URL router that maps urls to services and allows users to describe routes using regular expressions, evaluate and convert url paths into parameters.

Warning: To understand how the dispatcher router works, a little knowledge of regular expressions is required.

Whenever an Http request is received by the Dispatcher, a `\Comodojo\Dispatcher\Request\Model` object is created and hydrated with all the informations inside about the original request. This includes also:

- uri object representation;
- http method;
- attributes and parameters;
- headers;
- user agent.

Note: For more information about the request model, see request section.

Once ready, this object is used by the router to find the correct route to the requested service.

If there isn't any route that match with the request, a `\Comodojo\Exception\DispatcherException` is thrown, catched and forwarded to the output processor to create an HTTP (not found) response accordingly.

Note: For more information about the output processor, see the *Output Processor* section.

6.1 Anatomy of a route

All the routes that dispatcher can support are composed by 3 different parts:

- the base path;
- the variable path;

- the query string.

Except for the least, these are used to identify a service which is eventually invoked by the framework and initialized with the parameters extracted from the URI and the data (i.e. POST data).

Note: The HTTP schema and the FQDN (location) parts of the URI are not examined by the router, but can be accessed from the `\Comodojo\Dispatcher\Request\Model` object.

To represent the route, dispatcher uses a combination of JSON structures and regex expressions inside a route string.

Let's see an example of how routes can be defined.

```
routes/test/{"page": "p(\\d+)"}/{"ux_timestamp*": "\\d{10}", "microseconds": "\\d
↪{4}"}/{"filename*": "\\S+", "format*": "\\.(jpg|gif|jpeg|png)"}
```

We can split the route in two parts:

Base Path	Variable Path
routes/test/	{"page": "p(\d+)"}/{"ux_timestamp*": "\d{10}", "microseconds": "\d{4}"}/{"filename*": "\S+", "format*": ".(jpg|gif|jpeg|png)"}

The **routes/test** is the base path and is used to uniquely identify the service to invoke. You can add as many paths as you need: this is meant to be used to build a sort of hierarchical structure among the routes. For example, if you're building a framework which provides REST APIs, you can create routes with basic paths like "api/v2/users/add" or "api/v3/products/remove" and so on. This part is static and it must be presented at the beginning of the HTTP request (right after the http-schema+fqdn).

The variable path is instead defined by json strings which provide an association between parameter names and regular expressions used to identify them. Mandatory fields are marked with an asterisk "*" at the end; if a field neither mandatory nor presented to the router, it will be skipped and therefore it will not be available to the service.

The following urls can be intercepted by route above:

- routes/test/p15/1467727094/image.jpg
- routes/test/p4/14677270941234/test-case.png
- routes/test/1467727094/smile.gif?user=test

6.2 Attributes and Parameters

Attributes and parameters (including the querystring) are automatically processed by the router and placed into the `\Comodojo\Dispatcher\Request\Model` object. From the event or service perspective, these parameters can be accessed using the `Request::getQuery()` method.

Refferring to the previous example:

- The first parameter is called 'page' and, because it doesn't end with an asterisk, it's not required and can be omitted. When it's used, it must start with a 'p' followed by at least one number.
- The path that follows is composed by two different parameters, one of which (ux_timestamp) is required. This means that it must be part of the HTTP request and it have to be made of 10 digits. The second parameter tells you that you can add another 4 digits which will be accessible as "microseconds".
- The last path is similar to the previous, except that both parameters are required (they both end with an astersk).

The request urls shown in the previous chapter will call the service associated with the route "routes/test" which will receive the following parameters (represented here like a PHP array):

- *routes/test/p15/1467727094/image.jpg*

```
[
    "page" => array('p15', '15'),
    "ux_timestamp" => '1467727094',
    "filename" => 'image',
    "format" => array('.jpg', 'jpg')
]
```

- *routes/test/p4/14677270941234/test-case.png*

```
[
    "page" => array('p4', '4'),
    "ux_timestamp" => '1467727094',
    "microseconds" => '1234',
    "filename" => 'test-case',
    "format" => array('.png', 'png')
]
```

- *routes/test/1467727094/smile.gif?user=test*

```
[
    "ux_timestamp" => '1467727094',
    "filename" => 'smile',
    "format" => array('.gif', 'gif'),
    "user" => 'test'
]
```

Handling back-references

When a regular expression used in a route contains a back-reference, the parameter will be converted into an array where: - the first value is the full string; - the other values are the content of the back-references.

So, while *{"page": "p(\d+)"}* will lead to something like:

```
[
    "page" => array('p4', '4')
]
```

the same field (path) evaluated with *{"page": "p\d+"}* will lead to something like:

```
[
    "page" => 'p4'
]
```

6.3 Route definition

Every route can be defined using 4 different attributes:

- the route URL;
- the route type: *ROUTE*, *REDIRECT*, *ERROR* (see next sections);
- the class of the service to invoke (required only for *ROUTE* routes), in case of the endpoint of the route is a physical service;
- an array of parameters (optional), that can be used to configure optional - predefined - functionalities (e.g. route cache) or to extend them.

Route URL

The route URL is the complete representation of a route, as specified in the *Anatomy of a route* section.

Examples of valid routes are:

- api
- service/read
- page/get/{“page”: “\d+”}
- routes/test/{“page”: “p(\d+)”}/{“ux_timestamp*”: “\d{10}”, “microseconds”: “\d{4}”}/{“filename*”: “\S+”, “format*”: “.(jpg|gif|jpeg|png)”}

Service Class

This attribute defines the service that will be invoked by the router in case of a match. It has to be declared as a FQCN.

The service itself, shall be a valid service (*Writing services* section) that can be autoloaded.

Route parameters

The last attribute can be used to provide an array of parameters for the route. There is no limitation on the name or the type of a parameter. However, some special parameters are used to configure internal dispatcher features.

This pre-defined parameters are:

- **redirect-code**: used in case of a *REDIRECT* route to change the specify the HTTP code. By default, dispatcher will use 302, 303 or 307, depending on the case.
- **redirect-location**: the URL to redirect the client to.
- **redirect-message**: the message to include in the redirect content
- **redirect-type**: LOCATION (default) or REFRESH. The first uses HTTP redirect code to forward the client, the second creates a redirect page (200 OK Status Code) including the *Refresh* header and the redirect URI (in the page content).
- **error-code**: in case of an *ERROR* route, the error code to be used (default 500).
- **error-message**: the content of the HTTP error response (default *Internal Error*).
- **cache**: the service caching strategy, *SERVER*, *CLIENT* or *BOTH* (see *Enabling cache* for more information).
- **ttl**: the cache time to live, in seconds.

6.4 Route Installation

Routes can be installed in dispatcher in three different ways:

- programmatically;
- manually using a configuration file;
- automatically using the comodojo-installer package.

6.4.1 Add a route programmatically

In order to install a new route programmatically, the access to the `\Comodojo\Dispatcher\Dispatcher` object is required *before* invoking the `Dispatcher::dispatch()` method. Once gained, the main class can be used to get the router instance and then its routing table.

```
$dispatcher = new \Comodojo\Dispatcher\Dispatcher();

$router = $dispatcher->getRouter();

$table = $router->getTable();
```

In the routing table there are two methods that allow the installation of the route(s).

`Table::add()`

The `Table::add()` method can be used to install a single route:

```
$table->add(
    'routes/test/{"page": "p(\\d+)"}', // Route definition
    'ROUTE',                           // Route type
    '\\My\\Awesome\\Service',          // Service class
    [                                  // Parameters
        "cache" => "SERVER",
        "ttl"   => 3600
    ]
);
```

When you add a single route, this is volatile, it won't be stored in cache and the router won't remember it at the next startup.

`Table::load()`

This method is used to load one or multiple *permanent* route(s). The routes have to be passed as an array:

```
$table->load(
    [
        "route" => 'routes/timestamp/{"ux_timestamp*": "\\d{10}", "microseconds
↪": "\\d{4}"}',
        "type"  => 'ROUTE',
        "class" => '\\My\\Awesome\\TimestampService',
        "parameters" => []
    ],
    [
        "route" => 'routes/file/{"filename*": "\\S+", "format*": "\\.
↪(jpg|gif|jpeg|png)',
        "type"  => 'ROUTE',
        "class" => '\\My\\Awesome\\FileService',
        "parameters" => []
    ]
);
```

The routes added with this method will be stored in cache and will be reloaded at the next startup.

Note: The `Table::add()` method is meant to be used by plugins, that can interact with the router in a case-by-case manner, without persisting the modifications on the routing table into the cache.

`Table::load()`, instead, is designed to load a bunch of routes once and permanently (at least for the routing-table-cache ttl), and so it'is mostly useful in the framework startup. The comodojo/dispatcher project package, for example, adopt the following strategy to evaluate the router status and, in case, load the routing table from file:

```
if (
    file_exists($routes_file) &&
    empty($dispatcher->getRouter()->getTable()->getRoutes())
) {
    try {
        $routes = RoutesLoader::load($routes_file);
        $dispatcher->getRouter()->getTable()->load($routes);
    } catch (Exception $e) {
        http_response_code(500);
        exit("Unable to process routes, please check log: ".$e->getMessage());
    }
}
```

6.5 Bypassing Router

There are some cases in which the request, after being evaluated, should pass through the router only if a specific condition is met. If not, the request has to be redirected to a specific service or location (for example, to redirect an unauthorized request to the login service/page). This is also called *pre-routing bypass*.

To bypass the router, it is possible to create a plugin that install a listener to a pre-routing event, like the following one:

```php
<?php namespace My\Awesome;

use \League\Event\AbstractListener;
use \League\Event\EventInterface;

class RedirectToLogin extends AbstractListener {

    public function handle(EventInterface $event) {

        if ( $this->requestHasToBeReRouted($this->getRequest()) === false ) {

            $router = $event->getRouter();

            $route = new \Comodojo\Dispatcher\Router\Route();

            $route->setClass("\\My\\Awesome\\LoginService")
                ->setType("ROUTE");

            $router->bypassRouting($route);

        }

    }

    protected function requestHasToBeReRouted($request) {
        // some condition here //
    }

}

// a sample code to install the plugin
// $dispatcher->getEvents()->subscribe('dispatcher.request.#',
↪'\My\Awesome\RedirectToLogin');
```

6.6 Bypassing Service

In some other cases, afer a route has been found, the service should run only if a specific condition is met. This is also called *post-routing bypass*.

To skip the service, it is possible to create a plugin that installs a listener to a post-routing event and uses the `Router::bypassService()` method, like the following one:

```php
<?php namespace My\Awesome;

use \League\Event\AbstractListener;
use \League\Event\EventInterface;

class BypassSpecialService extends AbstractListener {

    public function handle(EventInterface $event) {

        if ( $this->serviceHasToRun($this->getRequest()) === false ) {

            $response = $event->getResponse();

            $response->getContent()->set("This service requires a special␣
↪authentication");
            $response->getStatus()->set(403);

            $router->bypassService();

        }

    }

    protected function serviceHasToRun($request) {
        // some condition here //
    }

}

// a sample code to install the plugin
// $dispatcher->getEvents()->subscribe('dispatcher.route',
↪'\My\Awesome\RedirectToLogin');
```

CHAPTER 7

The response model

Once the request is routed to an actual service, it is possible to compose a Response object. The router itself will execute the service and provide the resulting output to the Response objcet.

```
$response = new \Comodojo\Dispatcher\Response\Model(
    $router->configuration(),
    $router->logger()
);

$router->compose($response);

echo $response->content()->get();
```

7.1 Output Processor

7.1.1 The DispatcherException

CHAPTER 8

Writing services

8.1 Anatomy of a service

8.2 HTTP methods

8.3 Accessing framework APIs

8.4 Extra parameters

8.5 Enabling cache

CHAPTER 9

Plugins

9.1 How dispatcher emits events

9.2 Writing a plugin

9.3 Adding and enabling a plugin