rocketgraph

Report developer manual

Version 1.6a

1 What is "RAM"?

RAM stands for Report As a Modules, and is the engine that feeds reports to Rocketgraph. That means that each report in Rocketgraph's marketplace is a small program (module) that runs inside the main platform.
In that way, we provide a lot of helpful functionality, so the developer can focus on its data and the visualization.

2 Technical principles

Rocketgraph is created in PHP and supports only that language for report creation (for now…)
The version used is PHP 5.6.7, so this is the targeted version each report has to be implemented.
The templating engine we use is Twig, which can be used to create your views. This is just to help you create your report view and doesn't mean that this is the only way. You can use PHP files or the template engine of your preference, as long as it is provided within the report files. In short, the report has to return a valid HTML string. The tools that will be used are in developer's discretion
A Rocketgraph report consists of a main PHP class file, which is the entry point to the module. That class has to be an extension of RAM\BaseReport in order for the report to be implemented in the platform.
The provider's API keys are inserted in the report submit form, where the report connectors are selected.

3 Report folder structure

The minimum required file structure is the following:

- /report.php: Main file
- /public: Assets folder that will be publicly accessed (CSS/JS/images etc.)

Inside your report folder you can create as many folders as you want in order to use extra files and/or libraries

4 Main file structure

The entry point to your report is the main PHP class file. That file has to be an extension of RAM\BaseReport and be named exactly as the class name.
It also has to implement the render() method where all the report magic will happen and it will return the final report.

ex.

my_report.php

```
<?php
class my_report extends RAM\BaseReport
{

    public function render()
```

```
    {
        return 'Report content';
    }
}
```

This is pretty much the basic structure of a report. Below we will create our first report as an example.

5 Helper methods

Inside your report class you can find some methods that will help you utilize Rocketgraph's infrastructure and provider connection.

5.1 Connectors

The way you can access a connector and be able to request data from a provider's API, you can easily do that using the following methods.

5.1.1 getConnector(string $provider)

With this helper you can have access to the desired provider connector, which will be properly installed and ready to use. It takes as an argument the connector name. In case a connector that does not exist is asked, a *ConnectorNotFoundException* is thrown. So, a good practice is that you wrap that call in a *try/catch* block

ex.

```
try {
    $connector = $this->getConnector('twitter');
} catch (\Exception $ex) {
    return 'ERROR';
}
```

5.1.2 $connector->get(string $route, array $params = [], array $headers = [], $array = false, $useProxy = true, $permanent = false, $force = false)

When you retrieve a connector, via the *getConnector()* method, you can make calls to the according API, using the *get()* method of the connector.
Parameters:

- **$route** The API route
- **$params** Extra parameters for the route specified
- **$headers** Custom request headers
- **$array** If **TRUE** returns the result as an associative array
- **$useProxy** If **TRUE** the response is saved to a proxy storage for 1 day max
- **$permanent** If **TRUE** the response is saved permanently to the proxy storage and is never refreshed (needs $useProxy=true)
- **$force** If **TRUE** forces update the record in proxy storage even if it's permanent or not expired (needs $useProxy=true)

ex.

```
$response = $connector->get("users/show", array('screen_name'=>$handle,
'include_entities' => 'true'));
```

5.1.3 $connector->getLastHeaderS()

You can have access the the latest response headers by calling the connector's helper method getLastHeaders(). The response is an array of the connector's response headers.

ex.

```
$headers = $connector->getLastHeaders();
```

5.1.4 openConnector->get(string $url, $params = [], $array = false, $useProxy = true, $permanent = false, $force = false)

In case you want to use any open API that doesn't need any connection to user, you can use the embedded open connector that makes GET calls to any endpoint you define. This is very useful for generic APIs, like maps, weather etc.
Open connector's parameters are the same as any connector (see 5.1.2)

ex.

```
$response = $this->openConnector->get('http://my.apiendpoint.com');
$response = $this->openConnector->get('http://my.apiendpointwithparams.com',
['access_token' => 'my_access_token']);
```

5.1.5 sentiment->calculate(string $phrase)

Sentiment is our implementation of a sentiment analysis tool. This is a very handy tool and gives you the opportunity to make your reports even more interesting.
The sentiment analysis parses a given text and returns a float from -1 to 1 from the most negative to the most positive, with absolute neutral to be 0.

ex.

```
$sent = $this->sentiment->calculate('This a neither good or bad video'); //Returns 0.0
(neutral)
```

5.2 Render engine, scripts and styles

You can use our own rendering engine for HTML generation and script/style injection in the report headers. These are the helper methods you can use inside your report class.

5.2.1 addStyle(string $style)

This helper method will register your stylesheets for your report. The file path has to be relative to the public folder.

ex.

```
$this->addStyle('css/style.css');
```

5.2.2 addScript(string $script)

This helper method will register your JavaScript files for your report. The file path has to be relative to the public folder.

ex.

```
$this->addScript('scripts/script.js');
```

5.2.3 renderEngine->render(string $view, array $data)

The renderEngine parameter is the Twig engine that passes data in a view, renders its content and returns the HTML.

ex.

```
$content = $this->renderEngine->render('views/report.html', ['data'=>$data]);
```

5.2.4 View helpers

Rocketgraph's rendering engine is based on Twig and therefor it uses its syntax. Inside your view files, you can use some helpers like *path* to generate relative URLs for images or other files.

ex.

```
<img src="{{path('images/logo.png')}}" />
```

5.3 Custom rendering engine

In case you don't want to use our embedded rendering engine, the only rule you have to follow is the path of your assets. The path has to be relative and the same as the one passed to *path* helper

ex.

```
<img src="images/top_header.jpg"" />
```

5.4 Report parameters

Reports can have request parameters for any data needed by the end user in order for the report to be rendered.
These parameters can be a date, a date range, a number or anything else can be parsed through a form. All parameters are passed as a GET query parameter. For example, if I wanted to pass *handle* and *limit* the url will be appended like this:

?handle=myTwitterHandle&limit=10

All the necessary parameters are declared in the report submit form by the developer inside the **Input** box. If the report needs date range, then you check the "Show date range" and the according parameters are generated by Rocketgraph and are passed as *start_date* and *end_date* and use the format YYYY-MM-DD.

5.4.1 getParams()

With that method returns an array of all the available parameters that are filled by the user.

ex.

```
$params = $this->getParams();
```

5.4.2 getParam(string $parameter)

getParam returns the value of a parameter or *null* if the parameter does not exist.

ex.

```
$dateFrom = $this->getParam('date_from');
```

5.5 Storage

Reports can use out storage system in order to save calculated data and retrieve them when necessary in order to make more easy to create better report analytics.
The storage is accessible inside the report using the *storage* parameter:

ex.

```
$this->storage
```

5.5.1 Save

You can save data using the storage's *save* method. The storage is a key/value store engine and you can use as key and/or value any type of number, string or array.

ex.

```
$this->storage->save(['total_views', '2015-01-01'], 154);
```

The *save* method returns true if it succeeds or false if an error was occurred.

5.5.2 Find

You can retrieve any saved value by searching its key

ex.

```
$this->storage->find(['total_views', '2015-01-01']); //154
```

If the record does not exist, a *null* value will be returned.

5.6 Aggregations

For some reports there is the need of repeating calculating tasks in order to aggregate data and easily create comparisons with past values. For now these

aggregation tasks are once per day, but it will become more customizable in future.
In order to run an aggregation you 'll have to implement the *aggregate* method in your report. Whatever action is implemented in that method will be run once per day and with the use of report storage you can aggregate metrics that will be easy to use in the report render.

ex.

```
public function aggregate()
{
    $today = new \DateTime();
    $totalFollowers = $twitterParser->getCurrentFollowers();
    $this->storage->save([
        'total_followers',
        $today->format('Y-m-d') //2015-01-01
    ], $totalFollowers);
}
```

In this example we 've implement a method in our twitter parser that brings user's current followers.
You can retrieved that value using the same key:

```
$followers = $this->storage->find([
    'total_followers',
    '2015-01-01'
]);
```

6 Creating our first report

We will create a simple report, that uses Twitter API and renders the 3 top retweets from a profile.
We take as granted that the Twitter application that will be used in our report is properly registered and the correct key and secret is inserted upon report registration.

6.1 Create folder structure

Our folder structure will be the following:

- /twitter_report.php
- /model/TwitterParser.php (extra lib for tweet parsing)
- /views/report.html
- /public/css/style.css
- /public/images/top_tweets_bg.png
- /public/images/ribbon_1st.png
- /public/images/ribbon_2nd.png
- /public/images/ribbon_3rd.png

6.2 Create the main report file

Our main file is *twitter_report.php*. Inside that file we will include our library, register our CSS file and render library's results into our view.

twitter_report.php

```php
<?php
include 'model/TwitterParser.php';

class twitter_report extends RAM\BaseReport
{
    public function render()
    {
        $this->addStyle('css/bootstrap.min.css');
        $this->addStyle('css/style.css');
        $twitterConnector = $this->getConnector('twitter');
        $parser = new TwitterParser($twitterConnector);
        $data = $parser->getHandleData();
        return $this->renderEngine->render('views/report.html', array('data'=>$data));
    }
}
```

Our library that will help us process our data is TwitterParser, which is described here:

TwitterParser.php

```php
<?php
class TwitterParser
{
    protected $connector;

    public function __construct($connector)
    {
        $this->connector = $connector;
    }

    /**
     * Get the wanted data from user handle
     *
     * @return array
     */
    public function getHandleData()
    {
        $profile = $this->getProfile();
        $tweets = $this->getTweets($profile->screen_name);

        return $this->processTweets($tweets);

    }
```

```php
/**
 * Process tweets and get the desired data
 *
 * @param mixed $tweets
 * @return array
 */
protected function processTweets($tweets)
{
    $topRetweets = array();

    foreach ($tweets as $tweet) {
        if (isset($tweet->retweeted_status)) {
            /* Map top retweets */
            $topRetweets[$tweet->retweeted_status->retweet_count] = $tweet;
        }
    }

    /* Extract top retweets list and limit 3 top */
    ksort($topRetweets, SORT_NUMERIC);
    $topRetweets = array_slice(array_reverse($topRetweets), 0, 3);

    return [
        'top_retweets' => $topRetweets,
    ];
}

/**
 * Get latest tweets from handle
 *
 * @param string $handle
 * @return mixed
 */
protected function getTweets($handle)
{
    return $this->connector->get("statuses/user_timeline",
        array('screen_name'=>$handle));

}

/**
 * Get user profile info
 *
 * @return mixed
 */
protected function getProfile()
{
    return $this->connector->get('account/verify_credentials');
```

```
    }
}
```

6.3 Create the view

We will create 2 view file, just to demonstrate the use of template inclusion. For that case we will create the main *report.html* file and *top_tweets.html*. The second file we will place in the folder **partials.**

report.html

```
<div id="content">
    <div class="sub-section">
        {% include 'views/partials/top_tweets.html' %}
    </div>
</div>
```

partials/top_tweets.html

```
<img src="{{path('images/top_tweets_bg.png')}}" />
{% for tweet in data.top_retweets %}
<div class="top-tweet">
    <div class="block badge">
        {% if loop.index == 1 %}
        <img src="{{path('images/ribbon_1st.png')}}" />
        {% elseif loop.index == 2 %}
        <img src="{{path('images/ribbon_2nd.png')}}" />
        {% else %}
        <img src="{{path('images/ribbon_3rd.png')}}" />
        {% endif %}
    </div>
    <div class="block tweet-content">
        <div class="tweet-date col-xs-12">
            <div class="glyphicon glyphicon-calendar"></div>
            {{tweet.created_at|date("d.m.Y")}}
        </div>
        {{tweet.text}}
    </div>
    <div class="block retweets-block">
        <div class="count">
            {{tweet.retweeted_status.retweet_count}}
        </div>
        Retweets
        <div class="glyphicon glyphicon-retweet icon"></div>
    </div>
</div>
{% endfor %}
```

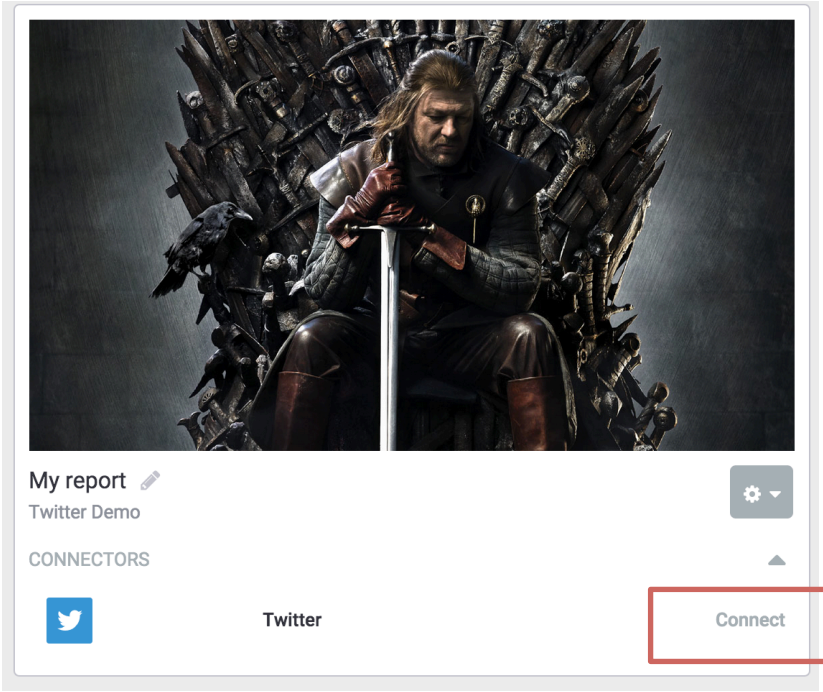
We will create a simple CSS file, just to arrange the data more proper

css/style.css

```
#content {
    width: 821px;
    height: 100%;
    margin: auto;
    background-color: #f3f3f5;
 }
.top-tweet {
   background-color: #FFFFFF;
   width: 90%;
   margin: 5%;
   float: left;
}
.top-tweet .block {
   padding: 3%;
   float: left;
}
.top-tweet .badge {
   width: 20%;
}
.top-tweet .tweet-content {
   width: 50%;
}
.top-tweet .retweets-block {
   width: 10%;
   text-align: center;
   margin-top: 3%;
}
.top-tweet .retweets-block .count {
   font-size: 2em;
}
```

6.4 Report Connection and render

Now, the report is ready to be used by any user that has a subscription to it. The user connects each report connector from its dashboard and views the result in its browser.



TOP TWEETS

1st	28.03.2015 RT @zend: [Infographic] 5 things you must know about #php7 http://t.co/XJaDBT3iFW #php	77 Retweets
2nd	09.01.2015 RT @DSpinellis: Είναι προαφανές ότι χώρα μας στοχάζεται πολύ σοβαρά το μέλλον της. http://t.co/n8Zz0A4xMu http://t.co/cHAmegELrI	26 Retweets
3rd	05.02.2015 RT @cnikitiadis: http://t.co/7wwwKIlFzs is looking for a talented, hands-on Senior Designer. Apply here: https://t.co/n0Tcu5XTZd	1 Retweets