Rest API Server Framework Documentation

Overview:

The framework is an instance of CodeIgniter 2.1 (http://codeigniter.com), with the Modular Extension (http://codeigniter.com/wiki/Modular_Extensions_-_HMVC) and the Rest Server library (https://github.com/philsturgeon/codeigniter-restserver) installed.

The Modular Extension was added to allow modularization of the code and so ensure the reusability.

The Rest Server library was installed to allow easy handle of the multiple REST methods (GET, PUT, DELETE, POST).

Regardless of the Modular Extension and the Rest Server library, all the features present in CodeIgniter remains the same. We still have a the MVC approach, and all CodeIgniter helpers, libraries, routing methods, and any other native feature can be used normally, so please take advantage of the great documentation available on http://codeigniter.com/user_guide/.

Requirements:

1. Apache web server
2. PHP 5.2 or higher
3. MySQL 5.1 or higher

Installing the framework:

1. Code checkout

The Framework in available at a SVN repository. You need to checkout the repository to get the code. You can do it by command line, but if you're using a Windows machine, use TortoiseSVN(http://tortoisesvn.net/downloads.html)

SVN repository info:

```
host:
http://apiframework.unfuddle.com/svn/apiframework_apiframework
username: developer
password: CpGUAI
```

2. MySQL Database creation

Under the folder ./docs in the SVN repository, there's a db_dump.sql file. Create a new database in your MySQL server and run the file against it.

3. Setting up CodeIgniter

Open the file ./app/application/config/database.php, and set up the hostname, username, password, and database entries.

4. Mod_rewrite

Finally, be sure your Apache server has mod_rewrite enabled. You can also set up a virtual host if you want to.

The document root must be the "app" directory. I've created two folders in the SVN repository, the "app" and the "docs". "Docs" will store, well, docs. "App" will store the code, so the "app" dir must be set as the document root of the website.

You don't have to set up virtual hosts, that's optional. If you don't want to set up a virtual host, don't use the .htaccess file at ./app/.htaccess, and on ./app/application/config/config.php, set the $config['index_page'] = 'index.php'. For example, let's say your file system looks like that:

```
- c:\
-- www (let's say this is your apache's document root)
--- apiframework
---- app
---- docs
```

To get to the system, you'll need to go to http://localhost/apiframework. Just remove the contents of the ./app/.htaccess and set the config.php file as I said above.

If you do want to set up a virtual host, add this lines to your httpd.conf file:

```
<VirtualHost *:80>
   DocumentRoot "c:/www/apiframework/app"
   ServerName "apiframework"
</VirtualHost>
```

And set up the hosts file so when you go to http://apiframework, it leads you to your local apache web server. Just open the file at c:\windows\system32\drivers\etc\hosts, and add this line:

```
127.0.0.1 apiframework
```

How to create an API:

Routing:

First of, you need to be aware of CodeIgniter's routing method. The URL relates with the code in a very straightforward way. For example, the URL

```
http://localhost/user/info/1
```

Will lead us to the following file:

```
- application
-- modules
--- user
---- controllers
----- user.php
```

The user.php file is our controller class. Note the url contains /user/info/1, so it will open the user.php file, create an instance of the "user" class, and call the "info" method, with the first argument being "1". The user.php file will look like that:

```php
<?php
class User extends CI_Controller
{
     function __construct() {
            parent::__construct();

            // Use this if you want this page be under HTTPS
            //force_ssl();
     }

     function info($user_id) {

            $this->load->model('user_model');
            $data['user'] = $this->user_model->get($user_id);
            $this->load->view('header');
            $this->load->view('user/info', $data);
            $this->load->view('footer');
     }
}
```

That's the standard CodeIgniter behavior. You can note the usage of the models and views, we will back to them later.

When creating an API server, we will NOT use the standard behavior, because we will be handling REST requests, but it is still very similar to that.

So if you want to create an API server function, first thing about the URL where it will be accessed. Lets say you want an API that returns a given user's information. A good URL for that would be

```
http://localhost/api/user/id/1
```

The file structure would be just like the standard CodeIgniter way, and we need to create those files and directories:

```
- application
-- modules
--- api
---- controllers
----- api.php
---- models
----- user_model.php
```

> Note: if you wonder where is the “views” directory, since we're creating an API server, the output will be either JSON or XML, and the Rest library handle the output of it. We just don't need the views.

Since we want to retrieve information, the REST method we will be using is GET. With all that in mind, the controllers/user.php file would look like that:

```
<?php
require(APPPATH.'/libraries/REST_Controller.php');

class Api extends REST_Controller
{
     function user_get()
     {
            if(!$this->get('id'))
            {
                   $this->response(NULL, 400);
             }

     // Note we're loading the user_model. I'll talk about it below.
     $this->load->model('user_model');
            $user = $this->user_model->get( $this->get('id') );

            if($user)
            {
                   $this->response($user, 200); // 200 being the HTTP
response code
            }
            else
            {
                   $this->response(NULL, 404);
            }
}
```

Note our REST method is GET. So the function to be called must have a "_get" suffix. Now, when doing a REST request, with method GET, to the URL http://localhost/api/user/id/1, the system will open the controller/api.php file, call the function user_get, and use the "id" parameter as an argument.

Database queries – using models

In the examples above, I mentioned the user_model.php file. Models are the layer that handles database queries, that's part of the MVC programming paradigm. The models gets into the folder... well, "models". The user_model.php file will look like that:

```
<?php

class User_model extends CI_Model {

    function __construct()
    {
                // Call the Model constructor
                parent::__construct();
    }

    function get_all()
    {
                $query = $this->db->get('user', 10);
                return $query->result_array();
    }

    function get($id)
    {
          $this->db->where('id', $id);
          $query = $this->db->get('user');

          return $row = $query->row_array();
    }
    function get_by_name($name)
    {
          $this->db->like('first_name', $name);
          $query = $this->db->get('user');

          return $query->result_array();

          // you can also use raw queries:
          $sql = "SELECT * FROM user WHERE first_name LIKE '%$name
%'";
          $query = $this->db->query($sql);

          return $query->result_array();
    }
}
```

The return value in the example above is an array containing all the rows resulting of the query.

Note I'm using the “Active Record” database abstraction class, but you can also use raw queries. I don't recommend it though. Using active record is not hard, adds security to your queries, and will allow you to preserve the model if the database engine changes. Read more at http://codeigniter.com/user_guide/database/index.html

So if you want to use a function from the model, first you need to load the model (i.e., instantiate the class), then call the methods from a controller. Just like we did in controllers/api.php code above.

API server response

We can give any output as a response for a request to our API. Lets say the client did not provided a required parameter, so we would include that in our code:

```
if (!$this->get('required_parameter'))
{
    $this->response('Bad Request. Please provide the required parameter',
400);
}
```

The code ends in that point and return the message we defined, so we can return any custom message and associated error code.

If everything goes well, we will should return that:

$this->response($result_array, 200);

The other REST methods

So far I used only GET method in the examples. But we also have PUT, POST, and DELETE. They are simple as GET, works in the same way. You just need to add the suffix you want to the method.

Lets say you want to delete an user record. So the request method is DELETE, the URL would be

```
http://localhost/api/user/id/1
```

In api.php controller file, we would add this method:

```
function user_delete()
{
     if (!$this->get('id')) {
          $this->response('Bad Request. Please provide the user ID', 400);
     }
     $this->load->model('user_model');
     $result = $this->user_model->delete($this->get('id'));
     if ($result) {
          $this->response('User deleted', 200);
     }
     else {
          $this->response('Database error', 400);
     }
}
```

Of course, you will need to implement the “delete” method in user_model.php, and make it to return true or false depending on how the delete query goes.

Security

The Rest library allows you to use a security method to authenticate users. Get to the file at ./application/rest.php.
The file is self-explanatory, the directives has comments above that explains what they do.

The framework is set up to use the “digest” authentication method, plus the usage of an API key, which is stored in the database table “keys”.
In the table “user”, we have an “key_id” field. That's just a sample of how we can use the APIs with the keys stored in the database. In a real system, we would have a code to generate random keys and relate then to users.

Each key may have an arbitrary level, and you can use that level to allow or restric access to a given function. For example,

```
function user_post() {
     if ($this->auth->level() < 5) {
          $this->response('Forbidden', 403);
     }
}
```

The function $this->auth->level refers to a method on a custom library, which is a normal CodeIgniter library. Read more at http://codeigniter.com/user_guide/general/creating_libraries.html

I'll explain this library a bit more in the API explorer section.

Logging system

All REST requests are recorded in the database table "logs". It stores the API key, the url, the client's IP, the method, the parameters, and the time when the request ocurred.

API explorer

An API explorer was added to the framework, it is a module under ./application/modules/explorer, so you can access it by going to a URL like http://localhost/explorer, or just going to the home page of your CodeIgniter website (see installing instructions at the top of this document). The API explorer is just a normal REST site, and you can define what parameters to use. The interface is simple and self-explanatory.

So our framework has to fronts, the REST server and the REST client, which is just a regular CodeIgniter website that makes uses of a rest client library.

In our CodeIgniter website, there is a custom "Auth" library, which handles user login and functions to retrieve the signed user's api key and level. It is autoloaded (see the file at ./application/config/autoload.php), so you can use them from anyplace in the system. For example:

```
<?php
echo 'api key: '. $this->auth->key().' - ';
echo 'level: '.$this->auth->level() ;
?>
```

Will print the signed user api key and level.

Creating new API clients

If you want to create an API client for a specific usage, do it as a regular CodeIgniter module, i.e., it must have a controller, it may have its own models or just use another module's models, and the view files.

To trigger a call to an API server, you can use the REST library, the CodeIgniter's cURL library, or yet PHP native cURL functions. Take a look at ./application/modules/explorer/controllers/explorer.php and check how to use them. There is an implementation of the rest library and a commented out section for native cURL usage.

Notes:

1. When you change the $route['default_controller'] = 'api'; that means going to the home page of the site will lead you to the respective controller.
   - The "API" controller is not a regular page, it is a REST service, so you can't just expect to open it as a regular website. It's an API that requests for a key.
   - If you don't want your API requiring a key, you can adjust the security settings on ./application/config/rest.php, as described in the "Security" section of the documentation.
   - The explorer function is the recommended way to test an API, cause it handles the request in the proper way. Just inserting the URL of the API into the browser is not the right way to do a request to a REST API (because the browser can't handle all the request headers and such).

2. So you created two sites, the first one is at http://localhost/apiframework. The other is athttp://localhost/restserver. The API explorer is set up to trigger requests only to http://localhost/apiframework. So I need to either change the API explorer or you need to put your functions under the http://localhost/apiframework

Notes:

If you don't want to modularize a function, you can use CodeIgniter's default directories. You can use the ./application/controllers, ./application/models, and ./application/views normaly.

You can also cross the modules. For example, lets say we can a ./application/models/log_model.php, and a module at ./application/modules/dashboard/controllers/dashboard.php. From the dashboard.php you can do:

```
$this->load->model('log_model');
```

If the system can't find a log_model.php under ./application/modules/dashboard/models/log_model.php, it will try to load from CodeIgniter's default directory, ./application/models.

If the model you want to load lives in another module, lets say an "user" module with a ./application/modules/user/models/user_model.php, you can do, from the dashboard.php file:

```
$this->load->model('user/user_model');
```

Finally, if you want to use blocks, i.e., calling a controller function from a view file, you can use that:

```
<?php echo Modules::run('module/controller/method', $param, $...); ?>
```

Creating light weight controllers

Validation / business rules layer

Implement the business rules on the controllers might increase the size of the code. CodeIgniter allows you to autoload validation rules automatically on each controller by using confiiguration files, so you don't need to create huge and complex validation functions within the controller. Read more at http://codeigniter.com/user_guide/libraries/form_validation.html#savingtoconfig.

Multiple controllers

If a controller or a module will have too many functions, you can split then into several sub-controllers instead of using a large one. For example, let's say you have several type of users to manage, each one with specific functions. You have "customer", "admin", "developer". You can use sub-controllers like that:

```
- application
-- modules
--- user
---- controllers
----- user.php (a file named "user.php" is not a requirement.)
----- customer.php
----- admin.php
----- developer.php
```

Then to call a request to a function under customer.php controller, the URL would be:

```
http://localhost/user/customer/[function_name]
```

Read more: http://codeigniter.com/user_guide/general/controllers.html#subfolders

References:

1. CodeIgniter: http://codeigniter.com/user_guide/index.html
2. Rest server library: https://github.com/philsturgeon/codeigniter-restserver
3. Rest server client: https://github.com/philsturgeon/codeigniter-restclient
4. CodeIgniter Modular Extension: https://bitbucket.org/wiredesignz/codeigniter-modular-extensions-hmvc/wiki/Home