

Monday, November 13. 2006

RESTful Web Services with Zend Framework

Introduction

We've all had over a week to absorb the new Zend Framework 0.20 release. Personally I've been loving it. The Zend Framework meets my requirements with flying colours.

For an upcoming project I intend implementing a RESTful web service. Back a few months ago when tinkering with the idea of a Technorati class for the Zend Framework, a reply from Davey Shafik highlighted the existence of a Zend_Rest_Client proposal. That proposal has since been implemented, and is now available from the Framework's incubator directory. Also available is the focus of this tutorial - Zend_Rest_Server.

So let's all gather round and do some tinkering. You can download the code put together here from [restful_tutorial.tar.gz](#) or [restful_tutorial.zip](#)

What is REST?

I'll start with the obligatory REST introduction.

REST stands for Representational State Transfer (see also [Representational State Transfer](#)). In an attempt to simplify understanding REST, consider the world wide web. Using the web, all resources can be requested using either a simple GET or POST request. Each resource has a unique URI. Additionally each resource can be cached since it's (for a period at least) static. It works really well for images, webpages and such.

Applying this concept to a web service means fitting all possible requests into a small number of possible URI resources. If you take an example of a Library Web Service, you may wish to request details for a specific book. As a webservice all we're looking for is the associated data for this book (as XML). A possible URI could be:

<http://www.example.com/book/isbn/xxx> or

<http://www.example.com/book?isbn=xxx> (using a query string)

Simply, a GET request to this URI would return the books data as XML as identified by the ISBN Number. What's important is that this URI always returns the same resource - it's cacheable. It might be updated in time, but the URI is persistent.

If we want the data for a book called "Ulysses", a possible URI could be:

```
http://www.example.com/book?title=Ulysses
```

Of course we could use the Zend Framework's parameter approach using a Zend Controller route. For good measure you can contrast this approach with XML-RPC (see [XML-RPC]) where requests are themselves XML containing a method name and sometimes parameter data, etc. Cutting out such complex XML requests, and returning to basic URI usage (GET, POST, PUT, DELETE) is the main difference between REST and RPC.

This tutorial takes a look at the new Zend_Rest_Server component. In doing so we implement a simple web service, and assess for advantages and disadvantages.

Application Setup

We'll start off by implementing a simple REST server using Zend_Rest_Server. The class is part of the Zend Framework which we'll use as the basis for our mini application. To start we need to setup a basic application using the Zend Framework. The directory structure we'll use is as follows:

```
restful-tutorial/  
  /application  
    /controllers  
    /servers  
  /library  
  /data
```

The Zend Framework itself has the `./library/Zend` directory placed in the application's `./library` directory. The Framework's `./incubator/library/Zend` directory is placed within the application's `./library/incubator` directory.

In the applications root directory we place the bootstrap `index.php` file and and a `.htaccess` file. The `index.php` is as follows.

```
/**
```

Bootstrap file for RESTful Web Service Tutorial

```
*/  
  
/*  
    The basics...  
    *  
    E_NOTICE not used since undefined variable notices  
    are reported from Zend_Server classes (see Issue  
    ZF  
    */  
    error_reporting((E_ALL|E_STRICT)& ~E_NOTICE);  
    ini_set('display_errors', 1); //disable on production servers!  
    date_default_timezone_set('Europe/London');  
  
/*  
    Setup the include_path to the ZF library  
    The core library classes take precedence  
    over the incubator classes  
    */  
    set_include_path(  
        './library' . PATH_SEPARATOR  
         . './library/incubator/library' . PATH_SEPARATOR  
         . './application/servers' . PATH_SEPARATOR  
    );  
    include 'Zend.php';  
  
/*  
    Use Zend::loadClass() to load essentials  
    */  
    Zend::loadClass('Zend_Config');  
    Zend::loadClass('Zend_Config_Ini');  
    Zend::loadClass('Zend_Controller_Front');  
    Zend::loadClass('Zend_Controller_RewriteRouter');  
/*  
    Grab database configuration  
    */  
    $config = new Zend_Config_Ini('./data/db.ini', 'local');  
    Zend::register('config', $config);  
  
/*  
    Setup RewriteRouter  
    */  
    $route = new Zend_Controller_RewriteRouter();  
  
/*  
    On my platform, I need to set the RewriteBase for ZF 0.20  
    RewriteBase is assumed to be $_SERVER['PHP_SELF'] after  
    removing the trailing "index.php" string  
    *  
    PHP_SELF can be user manipulated. Avoided using SCRIPT_NAME  
    or SCRIPT_FILENAME because they may differ depending on SAPI  
    being used.  
    */
```

```

$rewrite_base = substr($_SERVER['PHP_SELF'], , -9);
$route->setRewriteBase($rewrite_base);
/*
  @todo add new routes for future REST requests!
*/
/*
  Setup and run the Front Controller
*/
$controller = Zend_Controller_Front::getInstance();
$controller->setRouter($route);
$controller->run('./application/controllers');

```

Our .htaccess file includes:

```

RewriteEngine on
RewriteRule . index.php
php_flag magic_quotes_gpc off
php_flag register_globals off

```

Adding Population Data to Database

Being a web service, we need data to serve! To start I'm creating one table for MySQL with prepopulated data. We'll pretend our REST service allows users to retrieve notes originally stored by a Notelt application. There will be one table - "notes". Here's the SQL for a MySQL database you can import.

```

CREATETABLE`notes`(
  `id` int(11)NOTNULLAUTO_INCREMENT,
  `user` int(11)NOTNULLDEFAULT'0',
  `tag` varchar(32) character SET utf8 NOTNULLDEFAULT'notag',
  `text` varchar(255) character SET utf8 NOTNULL,
  PRIMARYKEY (`id`)
);
INSERTINTO`notes`VALUES(1, 1, 'php', 'Need to compile PHP 5.2.0!');
INSERTINTO`notes`VALUES(2, 1, 'security', 'Check INI options for ext/filter');
INSERTINTO`notes`VALUES(3, 2, 'ajax', 'Add JSON parser to extend String in chat_server.js');
INSERTINTO`notes`VALUES(4, 2, 'zend', 'Discuss bootstrap file with Lee re: error level reporting');
INSERTINTO`notes`VALUES(5, 2, 'php', 'Note in documentation - SimpleXMLElement::addChild()
requires PHP 5.1.3');

```

Adding URIs For Client Requests

Our web service will expect clients to make GET requests for this data. Oddly enough, users will not need all the data. To allow users append conditions to their requests, while maintaining a RESTful service we will design a number of possible urls to handle such conditions.

Rather than using a query string, it'll be difficult and rely on parameters set as part of the URI. These will be parsed from the URI by `Zend_Controller_RewriteRouter`.

The basic conditions we might expect to meet are retrieving notes based on:

user

tag

user and tag

id

We can create URI paths for these. In the example urls below, `service` stands for a Zend Framework `ServiceController` class, and `note` stands for a `NoteAction()` method on that Controller. The remainder of the url constitutes a parameter list.

`http://www.example.com/service/note/user/XXX`

`http://www.example.com/service/note/tag/XXX`

`http://www.example.com/service/note/user/XXX/tag/XXX`

`http://www.example.com/service/note/id/XXX`

We could also allow a client to grab all notes for all users, or all notes for all tags - but we'll restrict ourselves to the above possibilities. By keeping the possible URIs as simple as possible we make them easy to remember - since we're using the Zend Framework we can avoid using a query string and instead rely on the internal routing process which parses that part of the URI above our parent directory (which holds `index.php`) into controller, action and parameters.

The Zend Framework's core MVC components allow us to define and place requirements on additional routes using `Zend_Controller_RewriteRouter` and `Zend_Controller_Router_Route`. If we take the first URI, we can assemble a route of the form:

```
$routes = array();
$route['byuser'] = new Zend_Controller_Router_Route(
    / Set route format /
    ':controller/:action/user/:user',
    / Set applicable defaults /
    array('controller'=>'service', 'action'=>'book'),
    / Set variable requirements (Regex) /
```

```

    array('user'=>'ld+')
);

```

Using the above as a template we can quickly create new routes for each URI we intend supporting in our RESTful web service. We can replace the previous index.php @todo comment with the following:

```

/*
    @todo add new routes for future REST requests!
*/
$routes = array();
$routes['compat'] = $route->getRoute('default');
$routes['byuser'] = new Zend_Controller_Router_Route(
    ':controller/:action/user/:user',
    array('controller'=>'service', 'action'=>'note'),
    array('user'=>'ld+')
);
$routes['bytag'] = new Zend_Controller_Router_Route(
    ':controller/:action/tag/:tag',
    array('controller'=>'service', 'action'=>'note'),
    array('tag'=>'lw{3,32}')
);
$routes['byusertag'] = new Zend_Controller_Router_Route(
    ':controller/:action/user/:user/tag/:tag',
    array('controller'=>'service', 'action'=>'note'),
    array('user'=>'ld+', 'tag'=>'lw{3,32}')
);
$routes['byid'] = new Zend_Controller_Router_Route(
    ':controller/:action/id/:id',
    array('controller'=>'service', 'action'=>'note'),
    array('id'=>'ld+')
);
$route->addRoutes($routes);

```

ServiceController Class

Ignoring IndexController, we'll run our service from a NoteAction() method on a ServiceController class, i.e. URI after our root url will be /service/note.

With our valid routes defined, we can now implement the ServiceController class. The Controller will currently only require a single method - NoteAction(). The new Action will access the parameters parsed from the URI, and setup a Zend_Rest_Server instance to handle the current request.

The actual work entailed in gathering the data needed to build a response is offloaded on a standalone class we'll call App_Rest_Server_Note. In the absence of an application name, App_ is as good a class prefix as any. This class will hold all the specific methods Zend_Rest_Server may need to call.

The ServiceRestController (saved as ServiceController.php to ./application/controllers):

class ServiceController extends Zend_Controller_Action

```
{
    public function IndexAction()
    {
        / For anything other than a call to service/note
        redirect to IndexController
        */
        $this->_redirect('/');
    }

    public function NoteAction()
    {
        /*
        Fetch Parameters and Parameter Keys
        We don't need the controller or action!
        */
        $params = $this->_getAllParams();
        unset($params['controller']);
        unset($params['action']);
        $paramKeys = array_keys($params);

        /*
        Whitelist filter the Parameters
        */
        Zend::loadClass('Zend_Filter_Input');
        $filterParams = new Zend_Filter_Input($params);

        /*
        Build a request array, with method name to call
        on handler class for REST server indexed with
        'method' key.
        *
        Method name is constructed based on valid parameters.
        */
        $paramKeysUc = array();
        foreach($paramKeys as $key)
        {
            $paramKeysUc[] = ucfirst($key);
        }
        $methodName = 'getBy' . implode("", $paramKeysUc);
        $request = array(
            'method'=>$methodName
        );

        /*
        Filter parameters as needed and add them all to the
        $request array if valid.
        */
        foreach($paramKeys as $key)
        {
            switch($key)
```

```

    {
        case 'tag':
            $request[$key] = $filterParams->testAlnum($key);
            break;
        default:
            $request[$key] = $filterParams->testDigits($key);
    }
    if (!$request[$key])
    {
        // need better handling of filter errors for a real webservice...
        throw new Exception($request[$key] . ' contained invalid data');
    }
}

/*
  Setup Zend_Rest_Server
  Use App_Rest_Server_Note as handler
  */
require_once 'Zend/Rest/Server.php';
require_once 'App_Rest_Server_Note.php';

$server = new Zend_Rest_Server;
$server->setClass('App_Rest_Server_Note');
$server->handle($request);
}
}

```

The NoteAction() method isn't hugely complicated. It grabs an array of all parameters obtained from the URI. Once we have the parameters, we need to filter them since they are after all user input data. Once filtered, the filtered data is built into a new \$request array. We additionally add an extra 'method' entry to this array containing the method to call on the handler class.

With parameters filtered, we then instantiate a Zend_Rest_Server object and hand the \$request array to it. We also provide the Server instance with the name of the class it should use to handle the request and gather the data required for (a response).

There are some complications however. Our original assumption on the service would be that the parameters from the client can vary. Usually we could do something simple in the handler method on App_Rest_Server_Note like use func_get_args() to cope with variable numbers of parameters. Unfortunately, Zend_Rest_Server is very strict - you must have one public method for all possible parameter combinations on the handler class. In addition, the naming of such parameters must be identical to the names of the original parameters (i.e. see the variable requirements on our preset routes in index.php).

The only way to get around this, is adding additional code to compensate.

Before we go further, we also need a default IndexController class:

```
class IndexController extends Zend_Controller_Action
{
    public function IndexAction()
    {
        $this->_forward('Index', 'noRoute');
    }
    public function noRouteAction()
    {
        echo'The Notes web service is accessible from /service/note/.<br/>',
        'The URI requires additional parameters to be ',
        'passed appended to the URI in the form:<br/>',
        '/service/note/param1/value1<br/>The accepted parameters are: ',
        'tag, user, and id.';
    }
}
```

App_Rest_Server_Note Handler

To gather the data needed by Zend_Rest_Server to create a response, we will create an App_Rest_Note_Handler class. Because of the above limitations, we must have a specific public method for all possible parameter combinations. To avoid code duplication (our service only runs a simple SQL query with a varying WHERE condition), these simply delegate to a private getNotes() method.

The App_Rest_Server_Note class (saved as App_Rest_Server_Note to ./application/servers)

```
/*
    Server Class set in Zend_Rest_Server::setClass()
    *
    Since Zend_Rest_Server does not support variable parameter
    counts, I have to create specific public methods to handle the
    possible parameters to be passed. In addition the parameter
    * variable names must be identical to the variable name used
    when setting up these routes in the bootstrap index.php file.
    These all pass off the end task to the private getNotes().
    */
class App_Rest_Server_Note
{
    /*
        /service/note/user/xxx
        */
    public function getByUser($user)
    {
        return $this->getNotes(array('user'=>$user));
    }
    /*
        /service/note/tag/xxx /
    public function getByTag($tag)
    {
```

```

    return $this->getNotes(array('tag'=>$tag));
}
//service/note/user/xxx/tag/xxx /
public function getByUserTag($user, $tag)
{
    return $this->getNotes(array('user'=>$user,'tag'=>$tag));
}
//service/note/id/xxx /
public function getById($id)
{
    return $this->getNotes(array('id'=>$id));
}

/*
Use the passed parameters to build a WHERE
string for the necessary SQL query, run the
query and return the data for Zend_Rest_Server
to XMLify.
*/
private function getNotes(array$params)
{
    /*
    Create PDO class and connect to database
    */
    $config = Zend::registry('config');
    require_once 'Zend/Db.php';
    $dbParams = array(
        'host'=>$config->host,
        'username'=>$config->username,
        'password'=>$config->password,
        'dbname'=>$config->name
    );
    $db = Zend_Db::factory($config->type, $dbParams);
    /*
    Build WHERE part of the query from current
    parameters. This uses :key placeholders for
    binding values to the query string
    */
    $where = array();
    foreach($params as $key=>$value)
    {
        $where[] = $key . ' = :'. $key;
    }
    $whereString = implode(' and ', $where);

    /*
    Perform select query
    */
    $result = $db->query('select from notes where '. $whereString, $params);
    $data = $result->fetchAll();

    /*
    Since Zend_Rest_Server cannot handle multidimensional arrays

```

```

    like a set of results. We need to create a custom XML response
    using SimpleXML
    */
    return$this->getXML($data);
}
}

```

Once you get over the public method handoffs (the getNotes() method needs to know the name of parameters so it has both keys and values when building the SQL). The rest is simple. We run the query after establishing a database connection, and proceed to build an XML response.

Building an XML Response

It's here we find another Zend_Rest_Server limitation. First of all, the Server class can create an XML response itself, but it cannot yet handle a multidimensional array. To return the results, we therefore must create our own XML response. The referenced getXML() method for this follows.

```

private function getXML(array $data)
{
    /*
     * Create XML response.
     * Ignores special XML characters <>'"&
     */
    $xmlString = '<?xml version="1.0" standalone="yes"?><response></response>';
    $xml = new SimpleXMLElement($xmlString);
    $dataCount = count($data);
    for($i=0;$i<$dataCount;++$i)
    {
        $note = $xml->addChild('note');
        foreach($data[$i] as $key=>$value)
        {
            $note->addChild($key, $value);
        }
    }
    $xml->addChild('status', 'success');
    return $xml;
}

```

This method can build the expected response. If no data actually exists, we'll get an empty <response> element. In addition we add a <status> element for success. Zend_Rest_Server will use a similar element for any failures and include a failure message.

As of 0.20, Zend_Rest_Server does not support JSON, YAML or other formats for a response. We're restricted to XML until support for those formats are added.

Trying It Out

With all pieces in place, including the database (edit db.ini for your details) we can try a test url:

`http://yourserver/service/note/user/2`

The response resulting from this requests (just use your browser - we're not implementing a service client):

Posted by Pádraic Brady in PHP General at 07:26