Corey Maynard

Navigation

- Home
- Blog
- Portfolio
- About Me
- Contact Me
- Résumé

Search

Search

Archives

- 2015
- 2014
- 2013
- 2012
- 2011
- 2010
- 2009
- 2008

See it all!

RSS Feed

Subscribe to the RSS Feed for immediate updates and notifications.

Creating a RESTful API with PHP

17, May 2013

Contents

First, Some Background

What is REST?

REST, or in the full form, Representational State Transfer has become the standard design architecture for developing web APIs. At its heart REST is a stateless client-server relationship; this means that unlike many other approaches there is no client context being stored server side (no Sessions). To counteract that, each request contains all the information necessary for the server to authenticate the user, and any session state data that must be sent as well.

REST takes advantage of the HTTP request methods to layer itself into the existing HTTP architecture. These operations consist of the following:

- GET - Used for basic read requests to the server
- PUT- Used to modify an existing object on the server
- POST- Used to create a new object on the server
- DELETE - Used to remove an object on the server

By creating URI endpoints that utilize these operations, a RESTful API is quickly assembled.

What is an API

In this sense an API - which stands for Application Programming Interface - allows for publicly exposed methods of an application to be accessed and manipulated outside of the program itself. A common usage of an API is when you wish to obtain data from a application (such as a cake recipe) without having to actually visit the application itself (checking GreatRecipies.com). To allow this action to take place, the application has published an API that specifically allows for foreign applications to make calls to its data and return said data to the user from inside of the external application. On the web, this is often done through the use of RESTful URIs. In our cake example the API could contain the URI of

```
greatrecipies.com/api/v1/recipe/cake
```

The above is a RESTful endpoint. If you were to send a GET request to that URI the response might be a listing of the most recent cake recipes that the application has, a PUT request could add a new recipe to the database. If instead you were to request /cake/141 you would probably receive a detailed recipe for a unique cake. These are both examples of sensible endpoints that create a predictable way of interacting with the application.

Making Our Own RESTful API

The API that we're going to construct here will consist of two classes. One Abstract class that will handle the parsing of the URI and returning the response, and one concrete class that will consist of just the endpoints for our API. By separating things like this, we get a reusable Abstract class that can become the basis of any other RESTful API and have isolated all the unique code for the application itself into a single location.

However, before we can write either of those classes there's a third part of this that must be taken care of.

Writing a .htaccess File

A .htaccess file provides directory level configuration on how a web server will handle requests to resources in the directory the .htaccess file itself lives in. Since we do not wish to have to create new PHP files for every endpoint that our API will contain (for several reasons one of which being that it creates issues with maintainability); instead we wish to have all requests that come to our API be routed to the controller which will then determine where the request intended to go, and forward it on to the code to handle that specific endpoint. With that in mind, let's create a .htacccess file:

```
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule api/v1/(.*)$ api/v1/api.php?request=$1 [QSA,NC,L]
</IfModule>
```

What Did That Do?

Let's walk through this file. The first thing that we do here is wrap everything in a check for the existence of mod_rewrite.c; if that Apache module is present, we can continue. We then turn the RewriteEngine On and prepare it to work by giving it two rules. These rules say to perform a Rewrite if the requested URI does not match an existing file or directory name.

In the next line declares the actual RewriteRule. This says that any requests to api/v1/ that is not an existing file or directory should instead be sent to api/v1/index.php. The (.*) marks a named capture, which is sent along to the MyAPI.php script as well in the request variable through the use of the $1 delimiter. At the end of that line are some flags that configure how the rewrite is performed. Firstly, [QSA] means that the named capture will be appended to the newly created URI. Second [NC] means that our URIs are not case sensitive. Finally, the [L] flag indicates that mod_rewrite should not process any additional rules if this rule matches.

Constructing the Abstract Class

With our .htaccess file in place, it is now time to create our Abstract Class. As mentioned earlier, this class will act as a wrapper for all of the custom endpoints that our API will be using. To that extent, it must be able to take in our request, grab the endpoint from the URI string, detect the HTTP method (GET, POST, PUT, DELETE) and assemble any additional data provided in the header or in the URI. Once that's done, the abstract class will pass the request information on to a method in the concrete class to actually perform the work. We then return to the abstract class which will handle forming a HTTP response back to the client.

Firstly we're going to declare our class, its properties, and constructor:

```
abstract class API
{
    /**
     * Property: method
     * The HTTP method this request was made in, either GET, POST, PUT or DELETE
     */
    protected $method = '';
    /**
     * Property: endpoint
     * The Model requested in the URI. eg: /files
     */
    protected $endpoint = '';
    /**
     * Property: verb
     * An optional additional descriptor about the endpoint, used for things that can
     * not be handled by the basic methods. eg: /files/process
     */
    protected $verb = '';
    /**
     * Property: args
     * Any additional URI components after the endpoint and verb have been removed, in our
     * case, an integer ID for the resource. eg: /<endpoint>/<verb>/<arg0>/<arg1>
     * or /<endpoint>/<arg0>
     */
    protected $args = Array();
    /**
     * Property: file
     * Stores the input of the PUT request
     */
     protected $file = Null;

    /**
```

```
     * Constructor: __construct
     * Allow for CORS, assemble and pre-process the data
     */
    public function __construct($request) {
        header("Access-Control-Allow-Orgin: *");
        header("Access-Control-Allow-Methods: *");
        header("Content-Type: application/json");

        $this->args = explode('/', rtrim($request, '/'));
        $this->endpoint = array_shift($this->args);
        if (array_key_exists(0, $this->args) && !is_numeric($this->args[0])) {
            $this->verb = array_shift($this->args);
        }

        $this->method = $_SERVER['REQUEST_METHOD'];
        if ($this->method == 'POST' && array_key_exists('HTTP_X_HTTP_METHOD', $_SERVER)) {
            if ($_SERVER['HTTP_X_HTTP_METHOD'] == 'DELETE') {
                $this->method = 'DELETE';
            } else if ($_SERVER['HTTP_X_HTTP_METHOD'] == 'PUT') {
                $this->method = 'PUT';
            } else {
                throw new Exception("Unexpected Header");
            }
        }

        switch($this->method) {
        case 'DELETE':
        case 'POST':
            $this->request = $this->_cleanInputs($_POST);
            break;
        case 'GET':
            $this->request = $this->_cleanInputs($_GET);
            break;
        case 'PUT':
            $this->request = $this->_cleanInputs($_GET);
            $this->file = file_get_contents("php://input");
            break;
        default:
            $this->_response('Invalid Method', 405);
            break;
        }
    }
}
```

By declaring this an abstract class we're prohibited by PHP from creating a concrete instance of this class. From there we can create some protected class members. A protected member can only be accessed in the class itself and children thereof, unlike a private variable which can only be accessed in the class that defined the member.

Let's Talk About CORS

One of the core premises of an API is that clients on different domains than the one the API is hosted on will be connecting to the API to send and receive data. There is an inherit security risk here, as this can allow an attacker to create an imitation page and steal data sent back and forth. Therefore this ability must be explicitly enabled on pages that wish to allow what is called Cross-Origin Resource Sharing, aka CORS. One excellent resource to learn more about CORS is the website Enable CORS - it was quite helpful to me as I was trying to understand things.

For our API we need to make sure that this is enabled, so the very first thing that is done in the __construct method is to set some custom headers. The first two are the magic; firstly we allow requests from any origin to be processed by this page, next we allow for any HTTP method to be accepted.

Once the surprisingly simple yet completely crucial step of allowing CORS requests has been completed, it becomes time for our script to understand what the client has asked of it. To do that we're going to take the $request variable which will be sent to our script from the .htaccess file (remember? it contains the original

URI that the client requested), and tear it apart into the components we need. Once it's been exploded around the slash by pulling off the very first element we can grab the endpoint, if applicable the next slot in the array is the verb, and any remaining items are used as $args.

The HTTP method will describe the purpose of this request. GET requests are easy to detect, but DELETE and PUT requests are hidden inside a POST request through the use of the HTTP_X_HTTP_METHOD header. Once a method has been picked, the appropriate data source is parsed and cleaned for safety before being executed.

Completing the Abstract Class

The rest of the Abstract class comes next. Right now we're missing a function that will call the methods in the concrete class, and then one that will handle returning the response. Here's the rest of the abstract class:

```
abstract class API
{
    ...

    public function processAPI() {
        if (method_exists($this, $this->endpoint)) {
            return $this->_response($this->{$this->endpoint}($this->args));
        }
        return $this->_response("No Endpoint: $this->endpoint", 404);
    }

    private function _response($data, $status = 200) {
        header("HTTP/1.1 " . $status . " " . $this->_requestStatus($status));
        return json_encode($data);
    }

    private function _cleanInputs($data) {
        $clean_input = Array();
        if (is_array($data)) {
            foreach ($data as $k => $v) {
                $clean_input[$k] = $this->_cleanInputs($v);
            }
        } else {
            $clean_input = trim(strip_tags($data));
        }
        return $clean_input;
    }

    private function _requestStatus($code) {
        $status = array(
            200 => 'OK',
            404 => 'Not Found',
            405 => 'Method Not Allowed',
            500 => 'Internal Server Error',
        );
        return ($status[$code])?$status[$code]:$status[500];
    }
```

The one function worth mentioning here is the processAPI() method. This is the one publicly exposed method in the API, and its job is to determine if the concrete class implements a method for the endpoint that the client requested. If it does, then it calls that method, otherwise a 404 response is returned. The rest of the new code is simply an array map of all the possible HTTP codes and an input sanitizer.

That's all there is for the Abstract class. Now, finally time to implement a Concrete example.

Creating a Concrete API

Think back to our talk earlier about Cross-Origin Resource Sharing (CORS). Remember how it introduces a security vulnerability? We're going to work to close that as tightly as possible here by tying an Origin to a unique API Key. This means that only known and allowed external hosts will be able to connect to our API

service through a pairing of their domain name and a uniquely generated API Key. For the purposes of this example I'm going to leave some of the code to verify the API Key abstracted out. Additionally our API will require a unique token in every request to verify the User.

```
require_once 'API.class.php';
class MyAPI extends API
{
    protected $User;

    public function __construct($request, $origin) {
        parent::__construct($request);

        // Abstracted out for example
        $APIKey = new Models\APIKey();
        $User = new Models\User();

        if (!array_key_exists('apiKey', $this->request)) {
            throw new Exception('No API Key provided');
        } else if (!$APIKey->verifyKey($this->request['apiKey'], $origin)) {
            throw new Exception('Invalid API Key');
        } else if (array_key_exists('token', $this->request) &&
             !$User->get('token', $this->request['token'])) {

            throw new Exception('Invalid User Token');
        }

        $this->User = $User;
    }

    /**
     * Example of an Endpoint
     */
     protected function example() {
        if ($this->method == 'GET') {
            return "Your name is " . $this->User->name;
        } else {
            return "Only accepts GET requests";
        }
     }
 }
```

Using the API

Creating the concrete class is as simple as that. For each additional endpoint you wish to have in your API, simply add new functions into the MyAPI class whose name match the endpoint. You can then use the $method and $verb and $args to create flow paths within those endpoints.

To actually implement the API we need to create the PHP file that the .htaccess file is forwarding all of the requests to. In this example I named it api.php:

```
// Requests from the same server don't have a HTTP_ORIGIN header
if (!array_key_exists('HTTP_ORIGIN', $_SERVER)) {
    $_SERVER['HTTP_ORIGIN'] = $_SERVER['SERVER_NAME'];
}

try {
    $API = new MyAPI($_REQUEST['request'], $_SERVER['HTTP_ORIGIN']);
    echo $API->processAPI();
} catch (Exception $e) {
    echo json_encode(Array('error' => $e->getMessage()));
}
```

If you visit /api/v1/example (and have the User and Token system setup) you should see the output from that endpoint there.

That's a Wrap

That's all there is to it...which is, well, really quite a lot actually.

Comments

Mules said on May 18, 2013:

Great post! Off to make an API!....

Saul Goodman said on May 17, 2013:

It would be pretty awesome to integrate this example with some kind of authentication method for the user.

Corey Maynard said on May 19, 2013:

@Sauld - There's some hints to how I did that in there. The way I did it was using a unique token in the request. This token must be provided with every request to the API where a User is authenticated, and is only valid for a limited period of time. After which the user must re-authenticate either through password or cookies.

Corey Maynard said on May 30, 2013:

@Sauld - In the __construct of the MyAPI class I reference the $this->request['token'] which is a GET parameter. At that time you can do your timestamp verification by seeing if it is within the age window allowed.

Sauld Goodman said on May 21, 2013:

@Corey thanks for the answer. Lets say I replace the Api Key and user with temporal strings (for testing purposes)... how do I invoke the example method? Sorry, im not that used to oop. Thanks!

Krishna said on Sep 27, 2013:

Hi nice article, I have a small question. I am getting Put 405 method is not allowed error what is that mean? I am a front end developer no idea of server configuration. My mac osx has apache so I am using localhost as apache server.

Spoofie said on Sep 30, 2013:

Thanks for this great tutorial, this will help me alot.

alex said on Feb 14, 2014:

This is really, really helpful stuff, thanks a million.

Justin said on Apr 4, 2014:

Fantastic information! Thanks for sharing. This was a really thorough look at the basics of a REST API. I am looking into some best practices for the authentication piece and token generation. If you know of any good resources on that, I'd appreciate it.

Paulo said on May 18, 2014:

This is great! I can even use TCP to call the api and it doesn't breakdown, it handles everything (y)

Have Something to Say?

Questions? Comments? Concerns? Let me know what you're thinking.

- Your Name:

- Your Website (optional):
- Your Email (optional):
- Comments:

  You can use Markdown formatting here.

- What's 5 + 2?:
-

Submit

Recent Updates

Interested in what's been happening around here? Here's what I've had to say lately:

1. An Introduction to Bower – Nov 7, 2015
2. Performing AJAX POST Requests in Django – Jun 11, 2014
3. Introduction to Composer – Dec 14, 2013
4. Creating a RESTful API with PHP – May 17, 2013
5. Adding jQuery Event Listeners to Dynamically Created Django Admin Inline Model Fields – Apr 10, 2013

Not enough? Get your fix at the Archives. Or, subscribe to RSS feed.

Categories

All of my posts here fit under some category or another. Here are the top five categories:

1. Linux
2. PHP
3. Django
4. Design
5. SVN

You can find the rest of them at the Category List

About Me

I am Corey Maynard, web developer and designer in Baltimore, Maryland.

Here I write about my work, and topics that interest me in the field. Learn more About Me.