Web Services Tutorial

Lorna Mitchell

- PHP Consultant/Trainer
- API Specialist
- Speaker/Writer
- Open Source Contributor
- Twitter: @lornajane
- Site: `http://lornajane.net`

Agenda

- Web Services and a Simple Example
- HTTP and Data Formats
- Consuming Services from PHP
- Service Types and Soap
- RPC Services
- Building RESTful services

Theory and Introduction

What Is A Web Service?

- Means of exposing functionality or data
- A lot like a web page
- Integration between applications
- Separation within an application

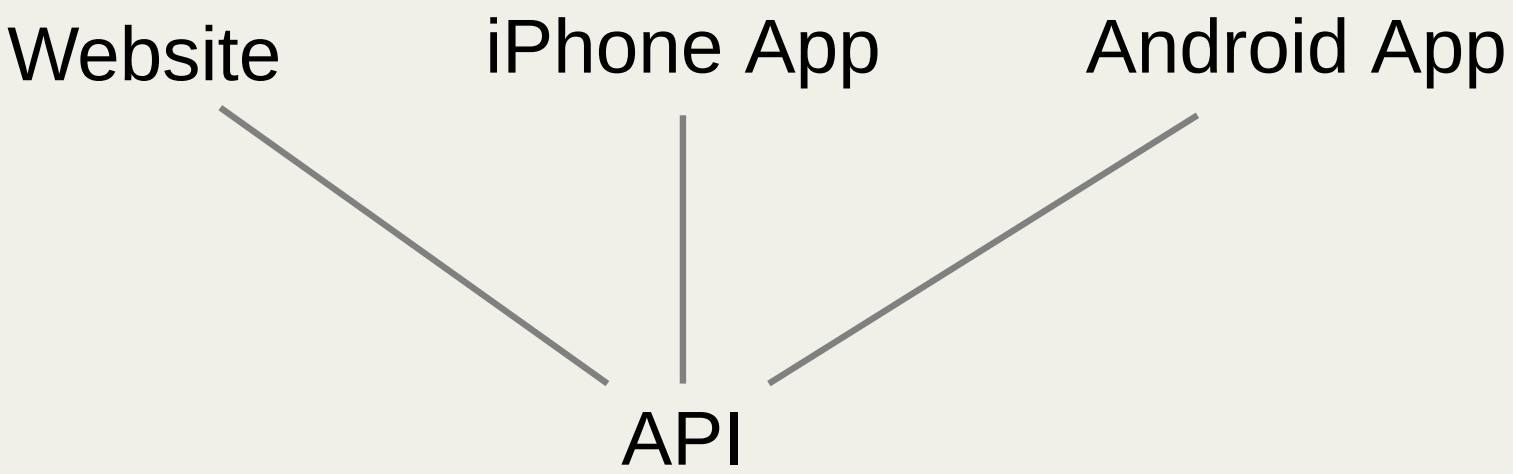
Web Services and You

This is not rocket science

You already know most of what you need!

Architecture Using APIs

Common to use an API internally as well as exposing it



Let’s Begin

Building Blocks

You can make an API from any tools you like

- Existing MVC setup
- Simple PHP (as in these examples)
- Framework module
- Component library

My First Web Service

- Make a virtual host
  - e.g. http://api.local
  - Don’t forget to restart apache
  - Add an entry to your hosts file

```
<VirtualHost *:80>
    ServerName api.local
    ServerAdmin admin@localhost
    DocumentRoot /var/www/myapi/public

    <Directory /var/www/myapi/public>
        AllowOverride All
        Order deny,allow
        Allow from All
    </Directory>
</VirtualHost>
```

api.vhost

My First Web Service

- Create the index.php file
  - e.g. /var/www/myapi/public/index.php

```php
$data = array(
    'format' => 'json',
    'status' => 'live'
    );
echo json_encode($data);
```

public/index.php

Consume Your Service

- **`curl http://api.local`**
- For more information about curl:
  - `http://curl.haxx.se/`
  - `http://bit.ly/esqBmz`

JSON JavaScript Object Notation

- Originally for JavaScript
- Native read/write in most languages
- Simple, lightweight format - useful for mobile
- In PHP we have **`json_encode`** and **`json_decode`**
- These work with arrays and objects

Our service returns:

```
{'format':'json','status':'live'}
```

Heartbeat Method

- A method which does nothing
- No authentication
- Requires correct request format
- Gives basic feedback
- Shows that service is alive

Delivering A Web Service

- Service
- Documentation
- Examples
- A help point

If you’re only going to build the service, don’t bother

HTTP and Data Formats

HTTP

HTTP is Hypertext Transfer Protocol - we'll recap on some key elements:

- Status Codes (e.g. 200, 404)
- Headers (e.g. Content-Type, Authorization)
- Verbs (e.g GET, POST)

Status Codes

A headline response. Common codes:

200	OK
302	Found
301	Moved
401	Not Authorised
403	Forbidden
404	Not Found
500	Internal Server Error

Consumers can get a WIN/FAIL indicator before unpacking the response in full

Working with Status Codes in PHP

We can observe status codes with curl, passing the -I switch

```
curl -I http://api.local
```

Let’s amend our web service, to return a 302 header

```
header("302 Found", true, 302);

$data = array(
    'format' => 'json',
    'status' => 'live'
    );
echo json_encode($data);
```

HTTP Verbs

- More than GET and POST
- PUT and DELETE to update and delete in a RESTful service
- HEAD, OPTIONS and others also specified

In REST, we use:

GET	Read
POST	Create
PUT	Update
DELETE	Delete

HTTP Headers

Headers are the metadata about the content we send/receive

Useful headers:

- **Accept and Content-Type**: used for content format negotiation
- **User-Agent**: to identify what made the request
- **Set-Cookie and Cookie**: working with cookie data
- **Authorization**: controlling access

Accept Header

What type of content can the consumer understand?

- **`-v`** with curl to see request and response headers
- **`-H`** to add headers

```
curl -v -H "Accept: text/html" http://api.local
```

Gives the output:

```
* About to connect() to api.local port 80 (#0)
*   Trying 127.0.0.1... connected
* Connected to api.local (127.0.0.1) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.21.0 (i686-pc-linux-gnu) libcurl/7.21.0 OpenSSL/0
> Host: api.local
> Accept: text/html
>
```

Using the Accept Header

We can work out what format the user wanted to see from the Accept header.

```php
$data = array(
    'status' => 'live',
    'now' => time()
    );

if(false !== strpos($_SERVER['HTTP_ACCEPT'], 'text/html')) {
    echo "<pre>";
    print_r($data);
    echo "</pre>";
} else {
    // return json
    echo json_encode($data);
}
```

public/headers.php

Content-Type Header

The Content-Type header: literally labels the contents of the response. We can include these in our examples:

```php
$data = array(
    'status' => 'live',
    'now' => time()
    );

if(false !== strpos($_SERVER['HTTP_ACCEPT'], 'text/html')) {
    header('Content-Type: text/html');
    echo "<pre>";
    print_r($data);
    echo "</pre>";
} else {
    // return json
    header('Content-Type: application/json');
    echo json_encode($data);
}
```

public/headers.php

Handling XML Formats

We can work with XML in PHP almost as easily

- Content type is text/xml or application/xml
- Two XML libraries in PHP
  - SimpleXML `bit.ly/g1xpaP`
  - DOM `bit.ly/e0XMzd`
- Give consumers a choice of formats

Adding XML to Our Service

```
$data = array(
    'status' => 'live',
    'now' => time()
    );

$simplexml = simplexml_load_string('<?xml version="1.0" ?><data />');
foreach($data as $key => $value) {
    $simplexml->addChild($key, $value);
}

header('Content-Type: text/xml');
echo $simplexml->asXML();
```

The result is this:

```
<?xml version="1.0"?>
<data><status>live</status><now>1302981884</now></data>
```

How to REALLY Handle Accept Headers

Example accept header (from my browser)

```
text/html, application/xml;q=0.9, application/xhtml+xml,
image/png, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
```

- See a much nicer example of this in **`headers-accept.php`**
- Taken almost entirely from the source of arbitracker
  - `http://arbitracker.org`
  - **`src/classes/request/parser.php`**
- Try this from curl, setting your own headers, and from your browser

Versions and Formats

- Always include a version parameter or media type
- Handle multiple formats, by header or parameter
  - JSON
  - XML
  - HTML
  - ?
- Common to detect header, allow parameter override

Statelessness

- Request alone contains all information needed
- No data persistence between requests
- Resource does not need to be in known state
- Same operation performs same outcome

Consuming Services from PHP

Consuming Your First Web Service

Three ways to consume web services:

- Via streams (e.g. file_get_contents for a GET request)
- Using the curl extension
  - `bit.ly/h5dhyT`
- Using Pecl_HTTP
  - `bit.ly/g4CfPw`

Using File_Get_Contents

This is the simplest, useful for GET requests

```
$response = file_get_contents('http://api.local/');
var_dump($response);
```

You can set more information in the stream context bit.ly/gxBAgV

Using Curl

This extension is commonly available

```php
$ch = curl_init('http://api.local/');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$response = curl_exec($ch);
var_dump($response);
```

Look out for the **CURLOPT_RETURNTRANSFER**; without it, this will **echo** the output

Using Pecl_HTTP

This is the most powerful and flexible, and can easily be installed from http://pecl.php.net

```
$request = new HTTPRequest('http://api.local/', HTTP_METH_GET);
$request->send();
$response = $request->getResponseBody();
var_dump($response);
```

Strongly recommend pecl_http if you are able to install pecl modules on your platform

Service Types

Web Service Types

There are a few types of web service

- RESTful
- RPC (Remote Procedure Call)
  - XML-RPC
  - JSON-RPC
  - Soap

RPC Services

These services typically have:

- A single endpoint
- Method names
- Method parameters
- A return value

A familiar model for us as developers

Soap

- Not an acronym
  - (used to stand for Simple Object Access Protocol)
- Special case of XML-RPC
- VERY easy to do in PHP
- Can be used with a WSDL
  - Web Service Description Language

Soap Example: Library Class

```php
class Library {
    public function thinkOfANumber() {
        return 42;
    }

    public function thinkOfAName() {
        return 'Arthur Dent';
    }
}
```

/public/soap/Library.php

Publishing a Soap Service

(don’t blink or you will miss it!)

```php
include('Library.php');

$options = array('uri' => 'http://api.local/soap');
$server = new SoapServer(NULL, $options);
$server->setClass('Library');

$server->handle();
```

/public/soap/index.php

Consuming a Soap Service

To call PHP directly, we would do:

```php
include('Library.php');

$lib = new Library();
$name = $lib->thinkOfAName();
echo $name; // Arthur Dent
```

Over Soap:

```php
$options = array('uri' => 'http://api.local',
    'location' => 'http://api.local/soap');
$client = new SoapClient(NULL, $options);

$name = $client->thinkOfAName();
echo $name; // Arthur Dent
```

Pros and Cons of Soap

- Many languages have libraries for it
- .NET and Java programmers in particular like it

Pros and Cons of Soap

- Many languages have libraries for it
- .NET and Java programmers in particular like it
- Weird things happen between languages regarding data types
- When it works, it’s marvellous
- When it doesn’t work, it’s horrid to debug (so I’ll show you how)

Pros and Cons of Soap

- Many languages have libraries for it
- .NET and Java programmers in particular like it
- Weird things happen between languages regarding data types
- When it works, it’s marvellous
- When it doesn’t work, it’s horrid to debug (so I’ll show you how)
- WSDL is complicated!

Working with WSDLs

WSDL stands for Web Service Description Language

- WSDLs were not designed for humans to read
- They are written in XML, and are very verbose
- If you do need to read one, start at the END and read upwards in sections
- Soap uses very strict data typing, which is an unknown concept in PHP

Read about WSDLs: `bit.ly/eNZOmp`

Debugging Soap

The Soap extension has some debug methods:

- Set the options to include trace=1 in the SoapClient
- **`getLastRequest()`**, **`getLastRequestHeaders()`**, **`getLastResponse()`**, **`getLastResponseHeaders()`** are then available

For all web service types, you can use:

- I like Wireshark (`http://www.wireshark.org`)
- Others use Charles (`http://www.charlesproxy.com`)
- If you like reading XML then use curl
- SoapUI (`http://soapui.org`)

Wireshark Demo

Bear with me while I fire up wireshark to show you

Extending Our Service

Consistency

- Important to retain
  - Naming conventions
  - Parameter validation rules
  - Parameter order
- Just as you would in library code

The Action Parameter

As a simple place to start, let's add an action parameter.

```
// route the request (filter input!)
$action = $_GET['action'];
$library = new Actions();
$data = $library->$action();
```

/rpc/index.php

Here's the code for the Actions class

```
class Actions {
    public function getSiteStatus() {
        return array('status' => 'healthy',
            'time' => date('d-M-Y'));
    }
}
```

/rpc/actions.php

So try http://api.local/rpc/index.php?action=getSiteStatus

Small APIs

- Beware adding new functionality
- Simple is maintainable
- Easy to use and understand

Adding an Action

To add a new action, create a new method for the Actions class, returning an array to pass to the output handlers

```
    public function addTwoNumbers($params) {
        return array('result' => ($params['a'] + $params['b']));
    }
```

/rpc/actions.php

But how do we pass the parameters in? Something like this

```
// route the request (filter input!)
$action = $_GET['action'];
$library = new Actions();
// OBVIOUSLY you would filter input at this point
$data = $library->$action($_GET);
```

/rpc/index.php

Access Control

A few common ways to identify users

- Username and password with every request
- Login action and give a token to use
- OAuth
- For internal applications, firewalls

RPC Service Example: Flickr

Take a look at http://www.flickr.com/services/api/

- Supports multiple formats (for request and response)
- Is well-documented and consistent
- Has libraries helping users to consume it
- Offers both RPC and RESTful (kind of!) interfaces
- Follows true XML-RPC (see http://en.wikipedia.org/wiki/Xml-rpc)

Vast numbers of applications using this API to provide flickr functionality elsewhere

RPC vs REST

We’ve seen an RPC service, but what about REST? The two are quite different

- RPC services describe protocols, e.g. XML-RPC
- RPC is a familiar: functions and arguments
- REST is a set of principles
- Takes advantage of HTTP features
- Typically has "pretty URLs", e.g. Twitter is mostly RESTful

RESTful Web Service

REST

- REpresentational State Transfer
- URLs are unique resource identifiers
- HTTP verbs indicate which operation should happen
- We have full CRUD operations on a series of resources

Here's one I prepared earlier: **`/public/rest/`**

REST as a Religion

Beware publishing a service labelled "RESTful"

- Someone will always tell you it isn’t
- They are probably right
- The strict rules around REST sometimes fit badly around business requirements
- Flamewars will ensue

Instead: "An HTTP Web Service"

Making Our Service Restful

We're aiming for URLs that look something like this:

- `http://api.local/rest/user`
- `http://api.local/rest/user/3`
- `http://api.local/rest/user/3/friends`

A specific item is a **resource**; where you request a list, e.g. **`/user`**, this is called a **collection**

All Requests to index.php

We want to push all requests through index.php, there is an .htaccess file that does this

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /

    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule ^(.*)$ index.php/$1 [L]
</IfModule>
```

/public/rest/.htaccess

Routing Within Our App

So our routing depends on the URL and on the verb used, so our controllers have a method per verb:

- GET /user
- Becomes UserController::GETAction()

```php
// route the request (filter input!)
$verb = $_SERVER['REQUEST_METHOD'];
$action_name = strtoupper($verb) . 'Action';
$url_params = explode('/',$_SERVER['PATH_INFO']);
$controller_name = ucfirst($url_params[1]) . 'Controller';
$controller = new $controller_name();
$data = $controller->$action_name($url_params);

// output appropriately
$view->render($data);
```

/public/rest/index.php

A Simple Controller

Now in UserController we add a GETAction

```
class UsersController {
    public function GETAction($parameters) {
        $users = array();
        // imagine retreving data from models

        if(isset($parameters[2])) {
            return $users[(int)$parameters[2]];
        } else {
            return $users;
        }
    }
}
```

/public/rest/controllers.php

Extending our Service

Next we will add something to get the user's friends

http://api.local/users/1/friends

```
class UsersController {
    public function GETAction($parameters) {
        $users = array();
        $friends = array();
        // imagine retreving data from models

        if(isset($parameters[2])) {
            if(isset($parameters[3]) && $parameters[3] == 'friends') {
                return $friends[(int)$parameters[2]];
            } else {
                return $users[(int)$parameters[2]];
            }
        } else {
            return $users;
        }
    }
}
```

/public/rest/controllers.php

Hypermedia

- The items returned are resources with their URLs
- This is **hypermedia** - providing links to related items/collections
- Allows a service to be self-documenting
- Allows a service to change its URLs - because they’re provided

Creating New Records

RESTful services use POST for creating data, so we have **POSTAction**

```
curl -X POST http://api.local/users -d name="Fred Weasley"
```

```
    public function POSTAction() {
        // sanitise and validate data please!
        $data = $_POST;

        // create a user from params
        $user['name'] = $data['name'];

        // save the user, return the new id

        // redirect user
        header('201 Created', true, 201);
        header('Location: http://api.local/rest/users/5');
        return $user;
    }
```

/public/rest/controllers.php

Updating Records

- To create: we used **POST**
- To update: we use **PUT**

The code looks basically the same, apart from:

```
// instead of this:
        $data = $_POST;

// use this:
        parse_str(file_get_contents('php://input'), $data);
```

Appropriate Status Codes for REST

- **GET**
  - 200 or maybe 302 if we found it somewhere else
  - 404 if we can’t find it
- **POST**
  - 201 and a location header to the new resource
  - or 400 if we can’t create an item
- **PUT**
  - 204 for OK (but no content)
  - 400 if the supplied data was invalid
- **DELETE**
  - 200 at all times, for **idempotency**

Responding to Failure

- Use expected response format
- Collate all errors and return
- Help users help themselves
- Be consistent
- Be accurate

Delivering Services

Technical Aspects

All PHP Best Practices apply equally to APIs

- Source control
- Unit/Integration testing
- Automated deployment
- Monitoring

Reliability and consistency are key

User Aspects

Who will use your API?

- Offer API Docs
- Write a quick start for the impatient
- Show examples, as many as possible
- Tutorials, blog posts, links to forum answers, anything
- Respond to questions

Questions?

Resources

All links are in the bit.ly bundle `bit.ly/emRpYT`

Thanks

- http://joind.in/talk/view/3338
- **@lornajane**
- http://lornajane.net