PHP Web Services, part 2

Last time: intro

- Using the JSON representation for data communicated across the Internet (XML is another way)
- How PHP makes it easy to convert from PHP arrays to/from JSON
- Web services using JSON on the wire
  - For ch05_guitar_shop, redone, to be working example
  - Ch05_guitar_shop, ch05_gs for short, now has client and server projects
    - Client: ch05_gs_client sends in web service requests
    - Server: ch05_gs_server answers web service requests
    - Uses Slim PHP component
  - Testing Web Services using command-line curl (not yet fully covered)

Provided ch05_gs_client/server

- Ch05_guitar_shop has been modified to use REST web services to access its data, to serve as a complete example for PHP web services.
- We’ll call it ch05_gs for short—

  ch05_gs_client------------------ch05_gs_server
  HTTP, JSON     PHP  HTTP, JSON     PHP

- Ch05_gs_client uses REST web services to access the database data, with the help of Guzzle.
  - Otherwise it’s the same as the old ch05_guitar_shop.
- Ch05_gs_server manages the database and answers the web service requests, with the help of Slim.
  - It has no user interface itself: it only answers web requests

Adding the browser to the picture: a client of gs_client:

Browser ------------ch05_gs_client------------ch05Gs_server
HTTP, HTML       PHP  HTTP, JSON     PHP

- So ch05_gs_client is both a server relative to the browser and a client relative to the REST server:
  - A server to the browser: user clicks a link, browser sends GET to /cs637/user/ch05_gs_client/product_manager, gets back HTML
  - A client to the REST server: ch05_gs_client needs a list of all toppings, sends GET to the REST server at /cs637/user/ch05_gs_server/api/toppings, gets back JSON

Slim component for REST servers

Minimal index.php for a Slim server: the “Hello World” of Slim...

```php
<?php
use \Psr\Http\Message\RequestInterface as Request;
use \Psr\Http\Message\ResponseInterface as Response;
require '../vendor/autoload.php';

$app = new \Slim\App;
$app->get('/hello/{name}', function (Request $request, Response $response, array $args) {
  $name = $args['name'];
  $response->getBody()->write("Hello, $name");
  return $response;
});
$app->run();
```

- If this file is at /service/api/index.php in the server, with access to the Slim code, a GET request to http://example.com/service/api/hello/Betty will answer “Hello Betty”
- See Slim Tutorial for full discussion of this code.

Slim server directory setup

Top-level dir (service here):

- composer.json: say we need php 5.5+, slim v3:
  ```json
  "require": {
    "php": "^5.5",
    "slim/slim": "^3.12"
  }
  ```
- composer tool (installed using apt-get on pe07)
  ```
  composer install downloads Slim v3
  ```
- vendor: directory of component code
  ```
  - Is created and filled by composer
    - api: directory with index.php, .htaccess
      - .htaccess: tell Apache webserver to route requests to index.php (requests with paths going to this directory, like /service/api/fooo)
    - This is all set up on pe07's apache server in /var/www/html/service
  ```
- Only needed to set up project, could delete after that.

```
Trying it out

pe07$ curl localhost/service/api/hello/Betty
Hello, Betty

pe07$ curl localhost/service/api/hello/Slim_works_great
Hello, Slim_works_great

To see important response info, use --i:

pe07$ curl --i localhost/service/api/hello/Betty
HTTP/1.1 200 OK
Date: Sun, 15 Nov 2020 20:30:05 GMT
Server: Apache/2.4.18 (Ubuntu)
Content-Length: 12
Content-Type: text/html; charset=UTF-8

Hello, Betty

Slightly bigger server

<?php
use \\Psr\\Http\\MessageServerRequestInterface as Request;
use \\Psr\\Http\\MessageResponseInterface as Response;
require '../vendor/autoload.php';
$app = new \\Slim\\App;
$app->get('/hello/{name}', getHello);
$app->post('/hello/poke', postPoke);

function getHello(Request $request, Response $response, array $args) {
    $name = $args['name'];
    $response->getBody()->write("Hello, $name");
    return $response;
}

function postPoke(Request $request, Response $response) {
    error_log('saw poke');
    $response->getBody()->write("OK");
    return $response;
}

But we want to read and write JSON data...

Idea of middleware

• All our responses need certain headers:
  - Content-type: application/json
  - Allow: 'GET, POST, PUT, DELETE' if we want to use more HTTP verbs than just GET and POST
  -Headers for CORS (cross origin resource sharing), so client browser can get HTML/PHP/JS from one server and this data from another

• The Slim middleware lets us specify this in one place and adds these headers on the way out. We just code (ignoring CORS here, see online code for that):

```php
$app->add(function ($req, $res, $next) {
    $response = $next($req, $res);
    return $response->withHeader('Content-Type', 'application/json')
                   ->withHeader('Allow', 'GET, POST, PUT, DELETE');
});
```

• The idea in general is that there is a chain of these functions, each fiddling with the response and then passing it to the next one. We are using only one such function.

• Note the chained method invocations: each method of $response returns the modified $response object.

PHP JSON Functions

• json_encode   Returns the JSON representation of a PHP value
• json_decode   Decodes a JSON string to PHP
• json_last_error Returns the last error

For a $product:

```php
$product_json = json_encode($product);
$product = json_decode($product_json, true);
```

REST Ch05_guitar_shop

REST API for project:
GET /categories
GET /categories/{category}/products
GET /categories/{category}/products/{pid}
POST /categories/{category}/products
DELETE /categories/{category}/products/{pid}

Examples:
GET /categories/guitars/products // get all guitars
POST /categories/guitars/products // add a guitar
DELETE /categories/basses/products/2 // delete bass 2

Now we consider how to implement these services

API implementation in brief

GET /categories
Get categories from DB, encode them in JSON array of objects and return it

GET /categories/{category}/products
Get products from DB for specified category, encode them in JSON array and return it

GET /categories/{category}/products/{pid}
Get product pid from DB, encode it in JSON object and return it

POST /categories/{category}/products
Get product in JSON from client, decode it and put it in the DB, return new URL in Location header

DELETE /categories/{category}/products/{pid}
Delete product pid in DB
Here we see two named placeholders, {cat} and {id}.

So we compute the new URL for the location, and attach it using another chained =

Example: GET to /cs637/user/pizza1/toppings/ is handled by

Of need to use

Database

Then we get what we really want, the parsed body, i.e. the PHP array
does the job, along with the addition of the

Note how

Since the incoming data of a POST request is in JSON, we would expect to

HTTP

but in fact Slim will do this for us…

Note

here

"Server

via a web server rule

Using HTTP

=== FALSE) {  // encode failed

json_decode

Here we add the product to the server's

Or maybe the client sent in the request with the wrong content

Request

Here the server couldn't encode its own database data: server error

encode error' .

and a POST to

'/categories/Basses/products/8' we would have

as the

in

Note the generality of "Parsed" here: the request must have the content

T

How can we get the web server to follow a new rule?

The server is blaming the client here, probably correctly

handling

in

json_decode

As "dot file", not listed by

Continuing in

The file

Bottom line: causes any request with local path starting with

•

•

•

•

•

•

•

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$app

$ap...
The default handling for exceptions in index.php code is set up at the beginning of the file: send back HTTP 500 with a JSON error message, and log the error.

Many functions, like getCategories, have no try/catch, just depend on the default setup.

But postProduct wants to blame the user for sending in a duplicate product (same productCode, causing the database insert to fail).

– So the postProduct code does try/catch on PDOException, checks for this case, handles it with HTTP 400 response, or if something else, does “throw $e” to hand it to the default handler, which then sends a HTTP 500

Sending back a server error should only happen when the server itself has a serious problem (i.e. bug). If the server is executing properly, it should send back a more descriptive code (why it couldn’t answer as expected)

- Can’t send an error page back, what do we do?
  – Log the error, set the response code in the HTTP response
  – Send the error message back inside a JSON message
- Choosing a good HTTP response code is important, since it is the primary way the client makes sense of what’s happening
- Codes used in this index.php:
  – 200 success
  – 500 server gave up/got exception it didn’t expect, takes the blame
  – 400 server couldn’t respond as expected but blames the client for sending in a bad request
  – We could use 201 Created for successful postProduct

<table>
<thead>
<tr>
<th>Status code summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number</strong></td>
</tr>
<tr>
<td>100-199</td>
</tr>
<tr>
<td>200-299</td>
</tr>
<tr>
<td>300-399</td>
</tr>
<tr>
<td>400-499</td>
</tr>
<tr>
<td>500-599</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status codes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number</strong></td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>301</td>
</tr>
<tr>
<td>303</td>
</tr>
<tr>
<td>400</td>
</tr>
<tr>
<td>401</td>
</tr>
<tr>
<td>403</td>
</tr>
</tbody>
</table>

\[\text{REST Web services: handling outgoing JSON}\]

<table>
<thead>
<tr>
<th><strong>Status codes (continued)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number</strong></td>
</tr>
<tr>
<td>404</td>
</tr>
<tr>
<td>405</td>
</tr>
<tr>
<td>414</td>
</tr>
<tr>
<td>500</td>
</tr>
</tbody>
</table>

With REST Web Services, JSON data is commonly returned from GET requests, and also from POSTs.

- It turns out to be easy: just “echo $jsonString;”
- That sends the contents of $jsonString to standard output, i.e. back to the client.
- We often need to set HTTP headers such as Content-type: application/json. That can be done by the PHP library function header(...)
- With Slim, we can use the method chaining methods of its Response object, like

  \[
  \$response->withHeader('Location', \$location)
  \rightarrow \text{withStatus(200)}
  \rightarrow \text{write(\$JSONContent)}
  \]

- Of course Slim is calling PHP’s header(...) and then echoing the contents.

With REST Web Services, JSON data is commonly returned from GET requests, and also from POSTs.

- It turns out to be easy: just “echo $jsonString;”
- That sends the contents of $jsonString to standard output, i.e. back to the client.
- We often need to set HTTP headers such as Content-type: application/json. That can be done by the PHP library function header(...)
- With Slim, we can use the method chaining methods of its Response object, like

  \[
  \$response->withHeader('Location', \$location)
  \rightarrow \text{withStatus(200)}
  \rightarrow \text{write(\$JSONContent)}
  \]

- Of course Slim is calling PHP’s header(...) and then echoing the contents.
REST Web services: handling incoming JSON

• With REST Web Services, JSON data is commonly POSTed to us (the server side), and we are expected to read it in and use it—how can we do this?? Or when using Slim, how does Slim’s code do it?
• As an exercise, you could try to find out from web resources—not so easy.
• Just as output to the client is standard output, input from the client needs standard input. How do we access it?
• The secret: ‘php://input’ is the filespec of standard input, i.e., the incoming data stream of the body of the request, and file_get_contents(‘php://input’) will get it all into a string. So that is how Slim is filling in $request for us.

Pizza2 REST web services

GET /pizza2_server/api/day returns the current day
POST /pizza2_server/api/day reinitialize DB, for testability
GET /pizza2_server/api/toppings returns info on all toppings
GET /pizza2_server/api/sizes returns info on all sizes
GET /pizza2_server/api/users returns info on all users
GET /pizza2_server/api/orders returns info on all orders
GET /pizza2_server/api/orders/{id} returns info on order of id (id)
POST /pizza2_server/api/orders adds an order
PUT /pizza2_server/api/orders/{id} updates an order

• Here “pizza2_server” is short for /cs637/username/pizza2_server
• You will implement this API using Slim

REST Resources

• Two kinds of resources:
  • day, a singleton, no collection involved
    – GET to …/day to read value, POST to reinitialize DB
  • Toppings, sizes, users, orders: normal collection resources
    – POST JSON to …/orders -> new order, say orders/12
    – New URI returned in Location header
    – But students don’t add toppings, etc., so only orders has POST
• GET to …/orders/12 gets JSON for order 12
• PUT to …/orders/12 updates order 12: used when a user acknowledges baked pizzas
• GET to …/orders gets JSON for all orders: used for listing order status for a user

Developing Web service code: notes

• PHP code in api/index.php of ch05_gs_server or pizza2_server
  o Web server code needs to avoid sending error text in response: will mess up other end’s interpretation
  o i.e., don’t “echo” debugging info: use error_log()
  o error_log() in web service code outputs to same file as the client side, so label output “client” or “server”, or use error_log only from the server side.
  o See slides 27-32 of Chapter 6 (6pp) for enabling and using error_log()
  o Actually you can use echo in server code if you test the server just using command-line curl, but final testing needs the client, so then you need to comment out all your echo statements.

REST web service client code

• The first job for client code is figuring out the url of the web services
• In our somewhat artificial setup, the server is a neighbor of the client on the same server:
  – …/user/pizza2_server/api
  – …/user/pizza2_client/model (for web_services.php)
• So the code in web_services.php drops ”/pizza2_client/model” off the end of the URL and adds ”/pizza2_server/api” to get $base_url for the web services
• This code also makes sure that the $base_url starts with http://localhost, so as to select the Apache server in XAMPP or pe07 on port 80, and not the Netbeans server on port 8383, since the Netbeans web server can’t interpret the crucial .htaccess file.
• In other more realistic setups, the base_url would be externally supplied.

Testing web services

• Web services (even old-style SOAP services) are “stateless”
• This means each service request contains all needed data for the server to do its job
• REST web services are just HTTP commands
• Therefore we can just fire HTTP commands at the server to test its services
• We can use command-line curl
We have looked at PHP’s `libcurl`, and the PHP component Guzzle, both of which can fire GETs and POSTs from our PHP code.

Separately, we can use curl at the command line in Windows or Linux/Mac.

More recently—installed Windows 10 systems have curl. See AddictiveTips post.

If you have an older Windows system, download curl for Windows at http://curl.haxx.se/download.html.

Linux/Mac: should have curl already.

Also see tutorial there: http://curl.haxx.se/docs/httpscripting.html.

**Command-line curl**

- We have looked at PHP’s `libcurl`, and the PHP component Guzzle, both of which can fire GETs and POSTs from our PHP code.
- Separately, we can use `curl` at the command line in Windows or Linux/Mac.
- More recently—installed Windows 10 systems have curl. See AddictiveTips post.
- If you have an older Windows system, download curl for Windows at http://curl.haxx.se/download.html.
- Linux/Mac: should have curl already.
- Also see tutorial there: http://curl.haxx.se/docs/httpscripting.html.

**Command-line curl example 1**

```bash
pe07$ curl
curl
```

This fires a POST to http://localhost/cs637/... With “9” in the POST body i.e. does the Web service to set the current day to 9 in the server, and has a Content-type header that says the POST body is text, -i option: i for “info” specifies display of response status code, response headers

Without -i:

```bash
pe07$ curl -d 9 -H Content-Type: text/plain
curl -d 9 -H Content-Type: text/plain
```

Nothing at all seen—how can we tell it worked?

With -i for verbose: see request headers, response status, headers, often too much output:

```bash
pe07$ curl -i localhost/cs637/eoneil/ch05_gs_server/api/categories
curl -i localhost/cs637/eoneil/ch05_gs_server/api/categories
```

**Command-line curl example 2**

```bash
pe07$ curl -i -d 9 -H Content-Type: text/plain
localhost/cs637/username/pizza2_server/rest/day
```

This fires a POST to http://localhost/cs637/... With “9” in the POST body i.e. does the Web service to set the current day to 9 in the server, and has a Content-type header that says the POST body is text, -i option: i for “info” specifies display of response status code, response headers

Without -i:

```bash
pe07$ curl -d 9 -H Content-Type: text/plain
localhost/cs637/username/pizza2_server/rest/day
```

Nothing at all seen—how can we tell it worked?

With -i for status info: less clutter, get the basic facts on the response:

```bash
pe07$ curl -i -H "Content-Type: application/json" -d @guitar.json
localhost/cs637/eoneil/ch05_gs_server/api/categories/products
```

Explanation of arguments:

- `-i` return response status, headers
- `-d @guitar.json` use method POST, with POST data from file guitar.json. Defaults to Content-Type for encoded parameters, like form data x=10&y=20
- `-H "Content-Type: application/json"` override the default Content-Type to this type, JSON

So this command does a POST to the URL with contents of guitar.json as POST data, and reports semi-verbosely on the action.
Shell scripts

We can automate command line work with shell scripts (even on Windows)

pe07$ more test1.sh
curl localhost/cs637/$1/ch05_gs_server/api/day
pe07$ chmod +x test1.sh
pe07$ test1.sh eoneil

Fills in eoneil for $1 in script:

Pw07$ curl -I -d 9 -H Content-Type:text/plain
localhost/cs637/eoneil/ch05_gs_server/api/day

For Windows: test1.cmd: use %1 instead of $1.
See shell and .cmd files in proj2_tests directory.

proj2_tests

- This directory is now available with 3 shell scripts for testing pizza2_server
  - Each with Windows and Linux/Mac versions
  - Will be used in grading run, but also useful for development

- Example: run server test1.sh on working pizza2_server project:

  pe07$ server test1.sh eoneil

  ------get server day: should show 1 if DB in init. state
  ------get toppings: returns toppings in JSON
  ------get sizes
  ------get users

server test1.sh

  echo -----------get server day: should show 1 if DB in init. state
curl http://localhost/cs637/$1/pizza2_server/api/day
echo
  echo -----------get toppings: returns toppings in JSON
curl http://localhost/cs637/$1/pizza2_server/api/toppings
echo
  echo -----------get sizes
curl http://localhost/cs637/$1/pizza2_server/api/sizes
echo
  echo -----------get users
curl http://localhost/cs637/$1/pizza2_server/api/users

server test1 eoneil fills in eoneil for each $1 in the above

server test1.cmd for Windows

rem -----------get server day: should show 1 if DB in init. state
curl http://localhost/cs637/1/pizza2_server/api/day
rem
rem -----------get toppings: returns toppings in JSON
curl http://localhost/cs637/1/pizza2_server/api/toppings
rem
rem -----------get sizes
curl http://localhost/cs637/1/pizza2_server/api/sizes
rem
rem -----------get users
curl http://localhost/cs637/1/pizza2_server/api/users
rem
rem

Same commands, just %1 for argument, rem for simple echo

Command-line curl example 3

pe07$ curl -i -H "Content-Type: application/json" -d @guitar.json
localhost/cs637/eoneil/ch05_gs_server/api/categories/guitars/products

HTTP/1.1 200 OK
Date: Tue, 17 Nov 2020 19:01:05 GMT
Server: Apache/2.4.18 (Ubuntu)
Location:
http://localhost/cs637/eoneil/ch05_gs_server/api/categories/guitars/products/13
Allow: GET, POST, PUT, DELETE
Content-Length: 116
Content-Type: application/json

{"categoryID":"1","productCode":"les_paul2","productName":"Gibson Les Paul2","listPrice":"1199.00","productID":"13"}

pe07$ more test1.sh
curl localhost/cs637/$1/ch05_gs_server/api/categories/guitars/products/13

pe07$ more test1.cmd
rem -----------get server day: should show 1 if DB in init. state
curl http://localhost/cs637/%1/pizza2_server/api/day
rem
rem -----------get toppings: returns toppings in JSON
curl http://localhost/cs637/%1/pizza2_server/api/toppings
echo
rem
rem -----------get sizes
curl http://localhost/cs637/%1/pizza2_server/api/sizes
echo
rem
rem -----------get users
curl http://localhost/cs637/%1/pizza2_server/api/users
echo
rem
rem

Same commands, just %1 for argument, rem for simple echo