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
• Proj 2 web services using JSON on the wire
  – For making supply orders, and finding out their status
  – For getting/setting the day in the server
• Proj 2 client and server projects
  – Server: proj2_server
  – Client: pizza2
• Testing Web Services using command-line curl
Project 2 supply orders

Product 11 is flour, 12 is cheese.
Order 40 units of flour and 60 units of cheese

$item0 = array('productID'=>11, 'quantity'=>40);
$item1 = array('productID'=>12, 'quantity'=>60);
$order = array('customerID'=> 1, 'items' => array($item0, $item1))
Same order in PHP and JSON

Array
{
    [customerID] => 1
    [items] => Array
        {
            [0] => Array
                {
                    [productID] => 11
                    [quantity] => 40
                }
            [1] => Array
                {
                    [productID] => 12
                    [quantity] => 60
                }
        }
    

    

}
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 this order:
  ```php
  $order_json = json_encode($order)
  $order = json_decode($order_json, true);
  ```
Use of `json_decode` in proj2_server

in handle_post_product($url) of /rest/index.php

$body = json_decode($bodyJson, true);

Here $bodyJson =

```
{"productID":"1","categoryID":"1","productCode":"strat","productName":"Fender Stratocaster","description":" The Fender Stratocaster is ... Tinted neck","listPrice":"699.00","discountPercent":"30.00","dateAdded":"2013-10-30 09:32:40","categoryName":"Guitars"}
```

So with the second-arg = true, we get a PHP associative array, instead of a “standard object”:

Array

```
{
    [productID] => 1
    [categoryID] => 1
    [productCode] => strat
    [productName] => Fender Stratocaster
    [description] => The Fender Stratocaster is ... Tinted neck
    [listPrice] => 699.00
    [discountPercent] => 30.00
    [dateAdded] => 2013-10-30 09:32:40
    [categoryName] => Guitars
}
```
Use of json_encode in proj2_server

In handle_get_product($url) of /rest/index.php

```
$product = get_product($product_id);  // get product from DB
$data = json_encode($product);  // convert AA to JSON
echo $data;  // send JSON back to client
```

Here $data =
```
{
"productID":"1","categoryID":"1","productCode":"strat","productName":"Fender Stratocaster","description":" The Fender Stratocaster is ... Tinted neck","listPrice":"699.00","discountPercent":"30.00","dateAdded":"2013-10-30 09:32:40","categoryName":"Guitars"
}
```

For product 1, but product 2 gets a null result, as we saw last time.

Debugging this: added, after json_encode:
```
error_log('json_last_error: '. json_last_error_msg());
```

Error log: json_last_error: Malformed UTF-8 characters, possibly incorrectly encoded
FYI: Finding bad chars in text

On UNIX, use the “od –c” command to display each byte as an ASCII char or, if it’s not an ASCII char, as a numeric code

Topcat$ od –c created.sql

...  
0010540 d e s i g n .   C u t t i n g -  
0010560 y e t - r i c h t o n e 342 200 224  
0010600 t h e   h a l l m a r k   o f  
0010620 t h e   L e s P a u l 342 200 224 p

• This shows 6 non-ASCII chars inside the product 2 data. Not all non-ASCII chars are also non-UTF8, but this is highly suspicious, and explains our problem with product 2
• Another way is using regex (regular expressions), covered in Chapter 15, which we skipped
Proj2 REST web services

Supplied web services

- POST /proj2_server/rest/day sets the current day number for the server.
- GET /proj2_server/rest/day returns the current day
- GET /proj2_server/rest/products/2 returns info on product 2
- POST /proj2_server/rest/products adds a new product

You will add

- POST /proj2_server/rest/orders creates a new supply order
- GET /proj2_server/rest/orders/123 returns info on order 123, including status (i.e., delivered or not)
- GET /proj2_server/rest/orders return info on all orders

Here “proj2_server” is short for /cs637/username/proj2/proj2_server
Resources

- Two kinds of resources:
  - day, a singleton, no collection involved
    - POST to .../day to change value
    - GET to .../day to read value
  - products, orders: normal collection resources
    - POST JSON to .../products -> new product, say products/12
    - POST JSON to .../orders -> new order, say orders/12
    - New URI returned in Location header
- GET to .../products/12 gets JSON for product 12
- GET to .../orders/12 gets JSON for order 12
- GET to .../orders gets JSON for all orders
Client and Server

• Server: proj2_server, derived from ch24_guitar_shop
  – Has web service code for the provided services
  – Has a new category for pizza supplies
  – You add web services for supply orders

• Client: pizza2, fixed-up pizza1 solution
  – You add inventory tracking, ordering supplies by using the web services
  – Has “restclient” code to get you started, exercises provided services
Logging, position-independence

- Both projects have the improved main.php
  - Include path set to project root using __DIR__
  - Error level set to debugging level
  - Error logging set up to output to local files
- Both projects can be moved around on the web server and still work
  - This involves using $app_path for internal URLs, and setting a robust include path
- Delivery is expected in
  /cs637/username/proj2/pizza2
  /proj2_server
REST Web service: the challenge of handling so many different URLs

• We have been writing server code all along.
  • Example: GET to /cs637/user/proj2/proj2_server/catalog/ is handled by /cs637/user/proj2/proj2_server/catalog/index.php via a web server rule

• Now want GET to .../proj2_server/rest/products/22 and POST to .../proj2_server/rest/orders and ... to be handled by .../proj2_server/rest/index.php

• How can we get the web server to follow a new rule?
• Answer depends on the web server: we’re using Apache
• The file rest/.htaccess does the job, along with the addition of the “rewrite module” and its configuration
  o As “dot file”, not listed by ls command in Linux/Mac
  o Need to use ls -a to see it
  o See its contents on server code handout: uses regex in rule
  o Bottom line: causes any request with local path starting with .../proj2_server/rest/ to be handled by proj2_server/rest/index.php
REST Web service: parsing the incoming URL

- OK, so now all the REST requests come in to this one index.php
- We can get the original URL using $_SERVER[REQUEST_URI]
- We can tell if it’s a GET or POST or whatever via $_SERVER[REQUEST_METHOD]
- The URL looks like …/proj2_server/rest/products/22
- We use string functions to pare it down to /rest/products/22
- Now we have the resource and the method (GET or POST), can handle each of the cases—see big switch ($resource) in code, with switch ($method) inside.
$project_uri = substr($request_uri, strlen($app_path) - 1);
$parts = explode('/', $project_uri);
// like /rest/product/1;
// 0 1 2 3
$server = $_SERVER['HTTP_HOST'];
$method = $_SERVER['REQUEST_METHOD'];
,proto = isset($_SERVER['HTTPS']) ? 'https:' : 'http:';
$url = $proto . '//' . $server . $request_uri;
$resource = trim($parts[2]);
$id = $parts[3];

switch ($resource) {
    // Access the specified product
    case 'products':
        error_log('request at case product');
    ...
}
REST Web service: the challenge of handling incoming JSON

• We can tell if it’s a GET or POST or whatever via $_SERVER[REQUEST_METHOD]

• GET requests are not so different from before: we just figure out what’s wanted from the URL and return it, in JSON.

• POST requests are different: before, the only data items that would ever come in a POST body were parameter values, and these were parsed out for us and put in $_POST.

• Now, JSON data is commonly POSTed to us, and we are expected to read it in and use it—how can we do this??
REST Web services: handling incoming JSON

- With REST Web Services, JSON data is commonly POSTed to us, and we are expected to read it in and use it—how can we do this??
- As an exercise, you could try to find out from web resources—not so easy.
- The secret: ‘php://input’ is the filespec of the incoming data stream, and file_get_contents(‘php://input’) will get it all into a string.
- You can see this inside the function handle_post_product on the code handout
- When the service creates a new resource, it returns the new URL in the Location header, along with status code 201: see code for this here too
function handle_post_product($url) {
    $bodyJson = file_get_contents('php://input');
    error_log('Server saw post data' . $bodyJson);
    $body = json_decode($bodyJson, true);
    try {
        $product_id = add_product($body['categoryID'],
                                  $body['productCode'],
                                  $body['productName'], ...
                                 )
                  // return new URI in Location header
                  $locHeader = 'Location: ' . $url . $product_id;
        header($locHeader, true, 201); // needs 3 args here
    } catch (PDOException $e) {
        $error_message = 'Insert failed: ' . $e->getMessage();
        include_once('errors/server_error.php');
        server_error(400, $error_message); // 400 = Bad Request
    }
}
Server errors/server_error.php

- Can’t send error *page* back, what do we do?
- Log the error, set the response code in the HTTP response
- Choosing a good response code is important, since it is the way the client makes sense of what’s happening

```php
function server_error($code, $error_message)
{
    error_log('Error in webapp , code ' . $code . ': ' . $error_message);
    http_response_code($code);
}
?>
## Status code summary

<table>
<thead>
<tr>
<th>Number</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-199</td>
<td>Informational</td>
<td>Request was received and is being processed.</td>
</tr>
<tr>
<td>200-299</td>
<td>Success</td>
<td>Request was successful.</td>
</tr>
<tr>
<td>300-399</td>
<td>Redirection</td>
<td>Further action must be taken to fulfill the request.</td>
</tr>
<tr>
<td>400-499</td>
<td>Client errors</td>
<td>Client has made a request that contains an error.</td>
</tr>
<tr>
<td>500-599</td>
<td>Server errors</td>
<td>Server has encountered an error.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
<td>Default status when the response is normal.</td>
</tr>
<tr>
<td>301</td>
<td>Moved Permanently</td>
<td>Requested resource has been permanently moved.</td>
</tr>
<tr>
<td>302</td>
<td>Found</td>
<td>Requested resource resides temporarily under a new URL.</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
<td>Request could not be understood by the server due to bad syntax.</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
<td>Request requires authentication. Response must include a www-authenticate header.</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
<td>Access to requested resource has been denied.</td>
</tr>
</tbody>
</table>
### Status codes (continued)

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>404</td>
<td>Not Found</td>
<td>Server could not find requested URL.</td>
</tr>
<tr>
<td>405</td>
<td>Method Not Allowed</td>
<td>Method specified in request line is not allowed for requested URL.</td>
</tr>
<tr>
<td>414</td>
<td>Request-URI Too Long</td>
<td>Typically caused by trying to pass too much data in a GET request. Usually resolved by converting the GET request to a POST request.</td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
<td>Server encountered an unexpected condition that prevented it from fulfilling the request.</td>
</tr>
</tbody>
</table>
REST Web service code: notes

- PHP code in rest/index.php of proj2_server
  - Web server code needs to avoid sending error text in response: will mess up other end’s interpretation
  - i.e., don’t “echo” debugging info: use `error_log()`
  - `error_log()` in web service code outputs to `php_server_errors.log` in the proj2_server directory.
  - `Server_error()` is called to send an error response. Need to choose the right status code.
  - Note the test scripts in proj2_tests. Use them to test your server code.
REST web service client code

- pizza2 has /restclient directory with client code for its services, just for testing
- This code should help you write needed code for the “real” client, the rest of pizza2
- You can execute this code by browsing to .../pizza2/restclient/ (whereas you can’t successfully execute .../proj2_server/rest this way)
- Last time we looked at a snippet of Guzzle code from there.
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:
  • …proj2/proj2_server/rest
  • …proj2/pizza2/restclient
• So the code drops “/pizza2/restclient” off the end of the URL and adds “proj2_server/rest” to get $base_url for the web services.
• In other more realistic setups, the base_url would be externally supplied.
REST Web Services in general

From **Dr Dobbs**:  
Addressing Resources  
- REST requires each resource to have at least one URI.  
- The job of a URI is to identify a resource or a collection of resources.  
- The actual operation is determined by an HTTP verb. The URI should not say anything about the operation or action.  
- This enables us to call the same URI with different HTTP verbs to perform different operations.  
- Our example: GET /.../orders vs POST /.../orders
Suppose we have a database of persons and we wish to expose it to the outer world through a service. A resource person can be addressed like this:

- http://MyService/Persons/1
- This URL has following format: Protocol://ServiceName/ResourceType/ResourceId
- Here are some important recommendations for well-structured URIs:
  - Use plural nouns for naming your resources.
  - Avoid using spaces as they create confusion. Use an _ (underscore) or – (hyphen) instead.
  - A URI is case insensitive. I use camel case in my URIs for better clarity. You can use all lower-case URIs.
    - This is wrong. The hostname part is case insensitive, but the path part is case sensitive on UNIX web servers because the filesystem is: try http://topcat.cs.umb.edu/cs637/eoneil/pizza1/DAY
  - You can have your own conventions, but stay consistent throughout the service.
Dr. Dobbs Example

- **http://MyService/Persons/1**
- Avoid verbs for your resource names until your resource is actually an operation or a process. Verbs are more suitable for the names of operations. For example, a RESTful service should not have the URIs http://MyService/FetchPerson/1 or http://MyService/DeletePerson/1.
Query Parameters in URIs

• Here is a URI constructed with the help of a query parameter:
  http://MyService/Persons?id=1
• The query parameter approach works just fine and REST does not stop you from using query parameters. However, this approach has a few disadvantages.
  – Increased complexity and reduced readability, which will increase if you have more parameters
  – Search-engine crawlers and indexers like Google ignore URIs with query parameters. If you are developing for the Web, this would be a great disadvantage as a portion of your Web service will be hidden from the search engines.
• The basic purpose of query parameters is to provide parameters to an operation that needs the data items. For example, if you want the format of the presentation to be decided by the client. You can achieve that through a parameter like this:
  • http://MyService/Persons/1?format=xml&encoding=UTF8
• Note: There is a better way to specify format and encoding using HTTP headers, covered later.
Links Between Resources

- A resource representation can contain links to other resources like an HTML page contains links to other pages.
- The representations returned by the service should drive the process flow as in case of a website.
- When you visit any website, you are presented with an index page. You click one of the links and move to another page and so on. Here, the representation is in the HTML documents and the user is driven through the website by these HTML documents themselves. The user does not need a map before coming to a website. A service can be (and should be) designed in the same manner.
- Let's consider the case in which a client requests one resource that contains multiple other resources. Instead of dumping all these resources, you can list the resources and provide links to them. Links help keep the representations small in size.
- Added by eoneil: However, following lots of links takes many round-trip times. We saw how much info was dumped by one video search...
Example with links: XML

Listing Six: A Club with links to Persons.

<Club>
  <Name>Authors Club</Name>
  <Persons>
    <Person>
      <Name>M. Vaqqas</Name>
      <URI>http://MyService/Persons/1</URI>
    </Person>
    <Person>
      <Name>S. Allamaraju</Name>
      <URI>http://MyService/Persons/12</URI>
    </Person>
  </Persons>
</Club>
First build PHP array, then encode it:

```php
$person1 = ['Name' => 'M. Vaqqas',
            'URI' => 'http://MyService/Persons/1'];

$person2 = ['Name' => 'S. Allamaraju',
            'URI' => 'http://MyService/Persons/12'];

$club = array('Name' => 'Authors Club',
              'Persons' => [$person1, $person2]);

json_encode($club, JSON_PRETTY_PRINT);
```
JSON for Club

{
   "Name": "Authors Club",
   "Persons": [
      {
         "Name": "M. Vaqqas",
         "URI": "http:\/\MyService\Persons\1"
      },
      {
         "Name": "S. Allamaraju",
         "URI": "http:\/\MyService\Persons\12"
      }
   ]
}
URI Templates

Dr. Dobbs example:
http://MyService/Persons/{PersonID}

For proj2_server:
Proj2_server/rest/orders/{orderId}
Content negotiation

• From Wikipedia:

Content negotiation is a mechanism defined in the HTTP specification that makes it possible to serve different versions of a document (or more generally, a resource representation) at the same URI, so that user agents (browsers, etc.) can specify which version fit their capabilities the best.

One classical use of this mechanism is to serve an image in GIF or PNG format, so that a browser that cannot display PNG images (e.g. MS Internet Explorer 4) will be served the GIF version.
Content Negotiation

• The user agent provides an Accept HTTP header that lists acceptable media types and associated quality factors.
• The server is then able to supply the version of the resource that best fits the user agent's needs.
• So, a resource may be available in several different representations. For example, it might be available in different languages or different media types, or a combination.
• For example, a browser could indicate that it would like to see information in German, if possible, else English will do. Browsers indicate their preferences by headers in the request. To request only German representations, the browser would send:
  
  Accept-Language: de

• Note that this preference will only be applied when there is a choice of representations and they vary by language.
Multiple preferences

As an example of a more complex request, this browser has been configured to accept German and English, but prefer German, and to accept various media types, preferring HTML over plain text or other text types, and preferring GIF or JPEG over other media types, but also allowing any other media type as a last resort:

Accept-Language: de; q=1.0, en; q=0.5
Accept: text/html; q=1.0, text/*; q=0.8, image/gif; q=0.6, image/jpeg; q=0.6, image/*; q=0.5, */*; q=0.1

- RFC 7231 does not specify how to resolve trade-offs (such as, in the above example, choosing between an HTML page in English and a GIF image in German).
XML vs. JSON

App 1 sends
Accept: application/json
Server sees that, sends response in JSON

App 2 sends
Accept: application/xml
Server sends XML
Languages

Browser 1 sends
Accept-Language: da (Danish)

Server sends HTML in Danish if possible
Accept-Language: da, en-gb;q=0.8, en;q=0.7

This means Danish is best for user, and English is of somewhat lower quality. Similarly the app can specify which image formats it wants.
Hypermedia Protocols

• If the links in the webservice results are well enough organized, a client can discover more pages and snake their way through related services.
• This is the basic idea of hypermedia.
• Example from REST in Practice, by Webber et al:
  – Customer POSTs order, response Order rep has link to cancel URI and make-payment URI
  – Customer POST to make-payment returns a Response that has link back to order and link to get-receipt
  – Customer GET to get-receipt returns a Response with link to order
  – Customer GET for Order (get-order) rep just has status (user needs to wait), no links
  – When order ready Order rep back from get-order has status=ready and link to receipt
  – When order done, Order rep has no links.
Recall from Chapter 22 slides...

The URL for the YouTube API documentation

http://code.google.com/apis/youtube/2.0/reference.html

The base URL for the YouTube Video Search API

http://gdata.youtube.com/feeds/api/videos

These are obsolete!
New base URL:

https://www.googleapis.com/youtube/v3/search

Note the https: protocol on the new URL: it needs a secure connection, our next topic. More info:

https://developers.google.com/youtube/
Some Available Web Services

Look at [Google APIs](https://developers.google.com/): all need https:
See translate, gmail, youtube, maps, etc., [PHP SDK](https://www.php.net) (Component)

[Amazon S3](https://aws.amazon.com/s3/): Storage API: see deprecated SOAP API, current REST API, has http:// endpoint, but requires https: for many actions [PHP SDK](https://www.php.net) (Component)

Facebook: [graph API overview](https://developers.facebook.com/docs/graph-api/), shows GET, POST, JSON snippet [PHP SDK](https://www.php.net) (Component)

Twitter: [REST API for tweets](https://developer.twitter.com/en/learning/how-to-use-the-twitter-api), [PHP component for REST](https://www.php.net)

[Video on REST APIs](https://www.youtube.com) with Facebook, Google Maps, Instagram examples