Project 2 supply orders

Order 40 units of flour (productID 11) and 60 units of cheese (productID 12)

From slides of Nov. 30, also expected in original servertest2:

```php
$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

```json
{
    "customerID": 1,
    "items": [
        {
            "productID": 11,
            "quantity": 40
        },
        {
            "productID": 12,
            "quantity": 40
        }
    ]
}
```

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
- **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

You added
- **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, **scalar resource**
  - 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

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
REST Web service: the challenge of handling so many different URLs

• How can we get the web server to use rest/index.php to handle a request to .../proj2_server/rest/products/22 ?
• 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 (need to use `ls -a` to see it)
  o See its contents, last line: uses regex pattern “^.*$” in rule
    `RewriteRule ^.*$ index.php [L]`
  o This 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

• .../proj2_server/rest/.htaccess ensures all the REST requests come in to this one index.php in the same directory
• 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
• Then we explode it into $parts: $parts[0] = ”,” $parts[1] = ”rest”, $parts[2] = ”products” (the resource), $parts[3]=”22” (the id)
• Now we have the resource and the method (GET or POST), can handle each of the cases—see big switch ($resource) in code, with switch ($method) inside.

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. More on file_get_contents(...) in Chapter 23.
• You can see this inside the provided function handle_post_product
• When the service creates a new resource, like a new product, it returns the new URL (.../rest/products/2 or whatever) in the Location header, along with status code 201, Created

REST Web service code: other notes

• PHP code in rest/index.php of proj2_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 `php_server_errors.log` (instead of `php_errors.log`)
• The provided curl scripts can be used to test/grade the server end.

Proj2_server: database use

• Use of old guitar shop tables:
  – Orders: holds orders submitted by pizza2
  – For new products: pizza and cheese, with rows in products (inserts in provided createdb.sql)
• New database table `systemDay` for holding server’s idea of current day (in provided createdb.sql)
  – model/day.php: new code to read and update the current day
GET orders/2 vs. GET orders

- Note there is common code needed for these two functionalities
- That is: turn a database-format order into a web-service-format order (with much less data)
- Also, interpret the delivery day to set the 'delivered' status.
- Ideally, this common code should be in its own function
- This function (get_order_data in the solution) is called from handle_get_order and from handle_get_orders.

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
- The browser can generate GETs for us: browse to http://topcat.cs.umb.edu/cs637/eoneil/proj2/proj2_server/rest/products/14
  - See JSON response:
    
    ```json
    {
        "productID": "14",
        "categoryID": "1",
        "productCode": "strat02",
        "productName": "Fender Stratocaster",
        "description": "The Fender Stratocaster... Tinted neck",
        "listPrice": "699.00",
        "discountPercent": "30.00",
        "dateAdded": "2017-04-30 15:01:40",
        "categoryName": "Guitars"
    }
    ```
- We can use command-line curl for GETs and POSTs

web service client code

- pizza2 has /restclient directory with client code for its web service requests, just for testing
- This code should help you write needed code for the “real” client, the rest of pizza2
- restclient has two PHP files:
  - index.php at top level, including util/main.php
  - scan.php: another app, FYI, not related to proj2
- index.php uses Guzzle to send off GETs and POSTs
  - Guzzle is installed in ../vendor, as usual for a component

Guzzle code in restclient/index.php

```php
// Instantiate Guzzle HTTP client
$client = new \GuzzleHttp\Client();
$url = 'http://' . $base_url . '/day/';
echo 'POST day = 3 to ' . $url . '<br>';
$error_log('...... restclient: POST day = 3 to ' . $url);
try {
    $response = $client->request('POST', $url, [
        'json' => 3]);
    $status = $response->getStatusCode();
} catch (\GuzzleHttp\Exception $e) {
    $status = 'POST failed, error = ' . $e;
    $error_log($status);
    include '../errors/error.php';  // Note new error.php code that handles Exceptions
}
```

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.

Guzzle code in restclient/index.php

```php
// Instantiate Guzzle HTTP client
$client = new \GuzzleHttp\Client();
$url = 'http://' . $base_url . '/day/';
echo 'POST day = 3 to ' . $url . '<br>';
$error_log('...... restclient: POST day = 3 to ' . $url);
try {
    $response = $client->request('POST', $url, [
        'json' => 3]);
    $status = $response->getStatusCode();
} catch (\GuzzleHttp\Exception $e) {
    $status = 'POST failed, error = ' . $e;
    $error_log($status);
    include '../errors/error.php';  // Note new error.php code that handles Exceptions
}
```

REST web service client code

- Once the base_url is known, the client uses Guzzle to generate GETs and POSTs, with methods for each request in admin/day/web_services.php.
  - It’s in the day directory because the supply orders are handled when the admin starts a new day. It could be in its own /restapi directory, say.
  - The next-day code needs to find out which supply orders have actually been delivered, vs. still undelivered, by querying the server using GETs.
  - The client could use the ability to request all the orders from the server, and find out that way, or use the saved undelivered order ids and query for each individually. It’s a design decision.
  - We still have to track undelivered orders to be able to detect newly-delivered ones, so you can credit the deliveries in the inventory.
Pizza2 inventory actions

- Needed tables, beyond pizza1:
  create table undelivered_orders {
    orderID integer not null,
    flour_qty integer not null,
    cheese_qty integer not null,
    primary key (orderID);
  }
  create table inventory {
    productID integer not null,
    productName varchar(255) not null,
    quantity integer not null,
    primary key (productID);
  }

  insert into inventory values (11,'flour', 100);
  insert into inventory values (12,'cheese', 100);

Day app

- Only two buttons, as before
  action=list or action = next_day in controller

List Orders: gather needed info

- if ($action == 'list') {
  try {
    $todays_orders = get_todays_orders($db, $current_day);
    $undelivered_orders = get_undelivered_orders($db);
  } catch (Exception $e) {
    include('../../errors/error.php');
    exit();
  }

  try {
    $inventory = get_inventory($db);
  } catch (Exception $e) {
    include('../../errors/error.php');
    exit();
  }

  include('day_list.php');

Credit newly delivered orders

- Two ways:
  - Use supply order ids from undelivered_orders and use GET /orders/{id} each of them for status.
  - If delivered, credit delivered quantities to inventory and delete id from undelivered_orders

- Or-

  - Get all orders from server (GET .../orders/)
  - Check each such delivered order for match with undelivered orders, and if so, credit, etc.

Change to next day code

- else if ($action == 'next_day') {
  try {
    finish_orders_for_day($db, $current_day); // old code
    $current_day = $current_day + 1;
    post_day($httpClient, $base_url, $current_day); // server: new day
    // credit non-delivered orders
    $undelivered_orders = get_undelivered_orders($db); // in helpers
    $supply = get_supply_orders($httpClient, $base_url); // from server
    record_deliveries($db, $supply, $undelivered_orders); // in helpers
    // Using updated inventory, figure out what to order, if anything
    $inventory = get_inventory($db);
    order_supplies($db, $httpClient, $inventory, $base_url);
  } catch (Exception $e) {
    include('../../errors/error.php');
  }

  // Redirect or set up variables for forward to day_list.php

Figure out what to order

- Need 150 units of flour in inventory or on order to keep going
- So if inventory_level < 150, compute 150 − inventory_level as needed_units, else 0.
- For flour, need number of 40-unit bags, but needed_units /40 is not an integer, and (int)(needed_units /40) is one too low most of the time, and (int)(needed_units/40) +1 is one too high for multiples of 40, though usable since we can order a little more than absolutely needed.
- What works neatly is order_qty = (int)(needed_units /40) + 1
- For cheese, use 20 instead of 40.
- Alternatively, can loop through possible order_qty values and compute their units until it's big enough. OK here, but not for large cases.