COMP519 Practical 18
REST (1)

Introduction

- This worksheet contains further exercises that are intended to familiarise you with Apache rewrite rules and the implementation of a RESTful web service using PHP.

  While you work through the exercises below compare your results with those of your fellow students and ask for help and comments if required.
- You might proceed more quickly if you cut-and-paste code from this PDF file. Note that a cut-and-paste operation may introduce extra spaces into your code. It is important that those are removed and that your code exactly matches that shown in this worksheet.
- The exercises and instructions in this worksheet assume that you use the Department's Linux systems to experiment with Apache rewrite rules and RESTful web services.
- To keep things simple, we will just use a text editor, a terminal, and a web browser. You can use whatever text editor and web browser you are most familiar or comfortable with.

Exercises

1. We want to develop a better understanding of Apache rewrite rules.

   a. Create the following directories:

   ```
   $HOME/public_html/rw
   $HOME/public_html/rw/aa    $HOME/public_html/rw/aa/bb
   $HOME/public_html/rw/cc    $HOME/public_html/rw/cc/dd
   ```

   b. Create a file `file1.html` in the directory `$HOME/public_html/rw/aa` containing the following HTML markup

   ```
   <!DOCTYPE html>
   <html lang='en-GB'>
     <head></head>
     <body>
       This is file <!--#echo var="document_uri"-->
     </body>
   </html>
   ```

   c. Make copies of `file1.html` in the directories `$HOME/public_html/rw/aa/bb`, `$HOME/public_html/rw/cc`, and `$HOME/public_html/rw/cc/dd`.

   d. Make sure that you have completed Practical 1 Exercise 7, that is, there is a world-readable file `.htaccess` in your `public_html` directory with the contents

   ```
   addOutputFilter INCLUDES .html .htm
   ```

   This enables the server-side include used in `file1.html`.

e. Use a web browser to check that all these files are accessible via the URLs

```
https://student.csc.liv.ac.uk/~<user>/rw/aa/file1.html
https://student.csc.liv.ac.uk/~<user>/rw/aa/bb/file1.html
https://student.csc.liv.ac.uk/~<user>/rw/cc/file1.html
https://student.csc.liv.ac.uk/~<user>/rw/cc/dd/file1.html
```

where *<user>* should be replaced by your University (MWS) username. Also check that each of these web pages correctly indicates it's location. That is, in the web browser you should see

```
This is file /~<user>/rw/aa/file1.html
```

for `https://student.csc.liv.ac.uk/~<user>/rw/aa/file1.html`, and, in analogy, corresponding paths for the other web pages.

f. Create a file `$HOME/rw/.htaccess` with the following content

```
RewriteEngine On
RewriteRule ^aa(\/?.*)$ cc$1
```

g. We almost always want to restrict access to files in our personal filestore, but `.htaccess` files need to be world-readable. Make sure that `$HOME/rw/.htaccess` is world-readable by using

```
chmod a+r $HOME/public_html/rw/.htaccess
```

h. Use a web browser to confirm that for the following two URLs nothing has changed:

```
https://student.csc.liv.ac.uk/~<user>/rw/cc/file1.html
https://student.csc.liv.ac.uk/~<user>/rw/cc/dd/file1.html
```

The part of those URLs subject to URL rewriting is `cc/file1.html` and `cc/dd/file1.html`, respectively. Since neither starts with `aa`, the pattern in the rewrite rule does not match and no rewriting takes place.

i. Use a web browser to access

```
https://student.csc.liv.ac.uk/~<user>/rw/aa/file1.html
```

In the web browser you should now see

```
This is file /~<user>/rw/cc/file1.html
```

indicating that the URL has been rewritten. This is because the part of the URL subject to rewriting is `aa/file1.html`, `^aa(\/?.*)$` does match `aa/file1.html`, `$1` in `cc$1` refers to the part matched by the capture group `(\/?.*)` which is `/file1.html`, resulting in the web server now looking for `cc/file1.html` instead. Note that the URL shown in the address/search field of the web browser has not changed.

j. Use a web browser to attempt to access

```
https://student.csc.liv.ac.uk/~<user>/rw/aa/bb/file1.html
```

In the web browser you should now see an error message, e.g., "This page isn't working". This is because the part of the URL subject to rewriting is `aa/bb/file1.html`, `^aa(\/?.*)$` does match `aa/bb/file1.html`, `$1` in `cc$1` refers to the part matched by the capture group `(\/?.*)` which is `/bb/file1.html`, resulting in the web server now looking for `cc/bb/file1.html` instead. This file does not exist. Note again that the URL shown in the address/search field of the web browser has not changed, so a URL that worked before no longer does

2. In the lectures we have discussed three different options how we can use rewrite rules in the implementation of a RESTful web service. We now know that rewrite rules work in principle, so let us see which of the three options will work for us.

   a. Replace the current content of `$HOME/public_html/rw/.htaccess` with the following:

   ```
   RewriteEngine On
   RewriteRule ^([ac].*)$ file2.php/$1
   ```

   b. Create a file `file2.php` in the directory `$HOME/public_html/rw/` containing the following HTML markup and PHP code

   ```
   <!DOCTYPE html>
   <html lang='en-GB'>
     <head></head>
     <body>
       <?php
         foreach (['REQUEST_METHOD','PATH_INFO','SCRIPT_FILENAME'] as $key)
           echo "\$_SERVER['$key'] => ",$_SERVER[$key],"<br>\n";
         foreach ($_REQUEST as $key => $value)
           echo "\$_REQUEST[$key] => $value<br>\n";
         $data = json_decode(file_get_contents('php://input'),TRUE);
         echo "<pre>";
         print_r($data);
         echo "</pre>";
       ?>
     </body>
   </html>
   ```

   c. In a web browser open the URL

   `https://student.csc.liv.ac.uk/~<user>/rw/file2.php`

   You should see

   ```
   $_SERVER['REQUEST_METHOD'] =>
   $_SERVER['PATH_INFO'] =>
   $_SERVER['SCRIPT_FILENAME'] => /home/<user>/public_html/rw/file2.php
   ```

   indicating that the environment variable `PATH_INFO` is not set, no query nor any additional data was passed to the script.

   d. Now open the URL

   `https://student.csc.liv.ac.uk/~<user>/rw/aa/bb`

   in the web browser. In all likelihood you just see an error message, e.g., “This page isn’t working”. The part `aa/bb` of the URL was subject to rewriting, the pattern `^([ac].*)$` matches it, in particular, the capture group matches all of `aa/bb`, and `file2.php/$1` becomes `file2.php/aa/bb`. The web server now looks for `file2.php/aa/bb`. Ideally, it would recognise that `file2.php` exists and pass `aa/bb` to that script via the environment variable `PATH_INFO`. But this is not the case.

   This means Option 2 discussed in the lectures does not work with the configuration of our web server.

   e. Replace the rewrite rule in `$HOME/public_html/rw/.htaccess` by the following one

```
RewriteCond %{REQUEST_URI} "!file2\.php"
RewriteRule ^([a-z].*)$ file2.php?resource=$1 [QSA]
```

and save the file. `RewriteCond` is not something we have discussed in the lectures. It ensures that the `RewriteRule` that immediately follows it is only applied if the condition of `RewriteCond` is true. The condition here is that `file2.php` does not already occur in the URL.

f. Reload the URL

`https://student.csc.liv.ac.uk/~<user>/rw/aa/bb`

in your web browser. You should now see the output

```
$_SERVER['REQUEST_METHOD'] => GET
$_SERVER['PATH_INFO'] =>
$_SERVER['SCRIPT_FILENAME'] => /home/<user>/public_html/rw/file2.php
$_REQUEST[resource] => aa/bb/
```

This means Option 3 discussed in the lectures works.

g. In your web browser access

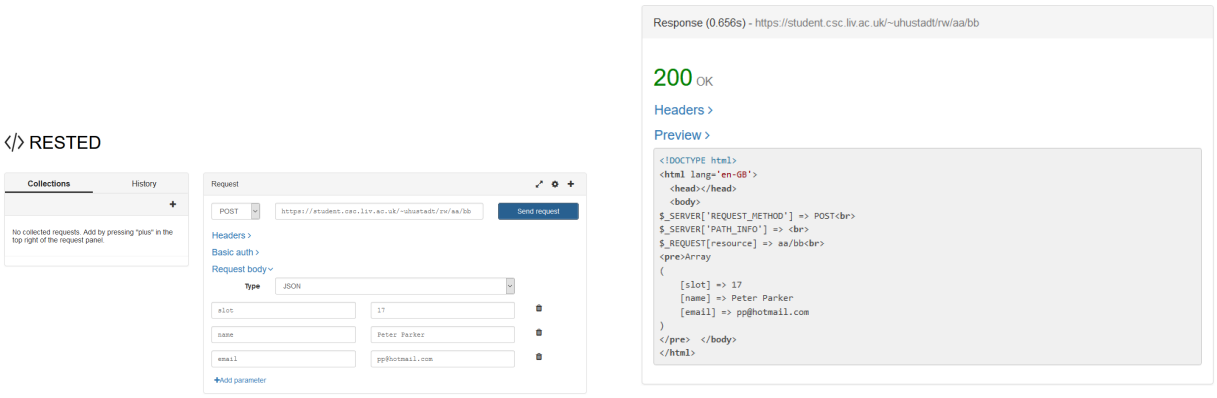
`https://student.csc.liv.ac.uk/~<user>/rw/aa/bb?sort=name`

Does the output shown to you change? If not, then `sort=name` has been lost. Try to fix the rewrite rule so that this query is preserved when rewriting the URL.

3. A web browser is sufficient to produce some GET HTTP requests, but to test a RESTful web service we need a better tool. One possibility is the Postman API Client [3]. Another possibility is the </> RESTED extension by Espen H for both Mozilla Firefox and Google Chrome [1, 2]. Install one of them.

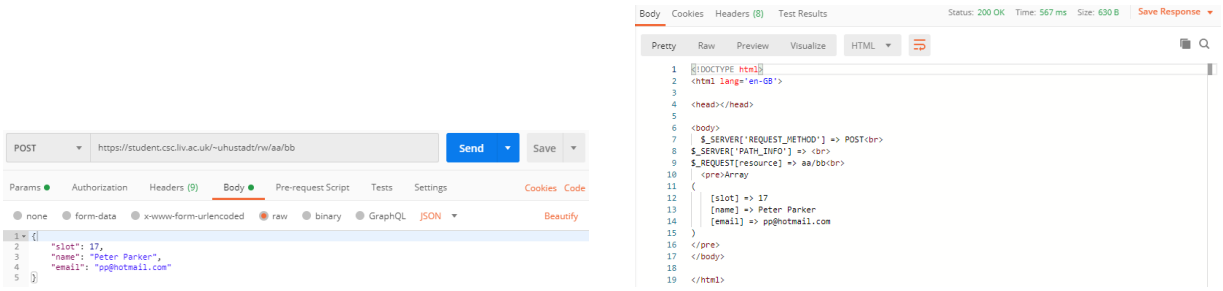
4. A REST API client allows you to generate arbitrary HTTP requests. They typically have a drop-down menu that allows you to select a HTTP request method and a input field for a URL. In addition, request headers and a request body can be specified.

   a. Let us first try to create a POST HTTP request.
   - Select 'POST' as request method in the drop-down menu.
   - Enter the URL



(a) POST Request (b) Response

Figure 1: </> RESTED Example



(a) POST Request (b) Response

Figure 2: Postman Example

`https://student.csc.liv.ac.uk/~`*`<user>`*`/rw/ee/ff`

where *`<user>`* should be replaced by your University (MWS) username.

- If you use `</> RESTED`, in Headers add the key 'Content-Type' with value 'application/json'.
- We want to send a request body that consists of data in JSON. Where that data consists of a single JSON object with a list of name/value pairs, `</> RESTED` allows to enter those in the same way as query parameter. Choose type 'JSON', then enter the names `slot`, `name`, and `email` with associated values 17, `Peter Parker`, `pp@hotmail.com`, respectively. See Figure 1a.
  In Postman, you need to click on 'Body', then select 'raw' in the list of buttons, and finally select 'JSON' from the drop-down menu of additional encodings. You can then enter arbitrarily complex JSON data into the text area. Enter

  ```
  {"slot":17,"name":"Peter Parker","email":"pp@hotmail.com"}
  ```

  See Figure 2a.
- Finally, click on the 'Send request' / 'Send' button to send the HTTP request.

b. If you have entered everything correctly, then a few seconds later you should see the response. See Figure 1b for what the response should look like in `</> RESTED` and Figure 2b for Postman.

   You can see that the HTTP response code is 200. The request URL has been rewritten and the PHP script `file2.php` has been executed, the HTTP request method we have used for our request can be seen in `$_SERVER['REQUEST_METHOD']`, `ee/ff/` ending up in `$_REQUEST['resource']`. The JSON data was accessed by `file2.php` via `php://input` and decoded into an associative array.

c. One of the reason for using tools like `</> RESTED` and Postman is the ease with which we can create HTTP requests with a range of request methods. You have already used the drop-down menu for the request method to create a POST request. Now systematically create GET, PATCH, and DELETE requests. Do not change any of the other settings. After sending these requests observe what the response looks like.

   You should see that the value of `$_SERVER['REQUEST_METHOD']` reflects the HTTP request method that you have chosen, but nothing else changes. This might be a bit surprising as, for instance, a GET request is not expected to have a request body. But it turns out that the web server and PHP are agnostic about that.

d. So far our PHP script always produces an HTTP response code 200. We want to change that. Add the following code at the end of the PHP script in `file2.php`.

   ```
   $status = ['GET' => 200, 'DELETE' => 204, 'PATCH' => 404, 'POST' => 201];
   http_response_code($status[$_SERVER['REQUEST_METHOD']]);
   ```

Make sure that you have not introduced any syntax errors into the code.

e. Now, systematically send DELETE, GET, PATCH, and POST requests as you have done in Exercise 4c. Observe how the response shown in </> RESTED or Postman differs between the four request methods.

5. `file2.php` is typical for our PHP scripts so far. It's an HTML document with PHP code embedded. This makes sense since our focus had been on PHP scripts that produce HTML documents as 'output' (dynamic web pages), or, alternatively, to add responsiveness to HTML documents.

   However, in the final part of the module our focus is on RESTful web services. Their output is meant to consist of an HTTP response code and JSON or XML data. Let us try to create a simple PHP that does so.

   a. Using a text editor, create a file `file3.php` in the directory `$HOME/public_html/rw/` with the following content

```
<?php
$status = ['GET' => 200, 'DELETE' => 204, 'PATCH' => 404, 'POST' => 201];
$output = $_REQUEST;
$output['REQUEST_METHOD'] = $_SERVER['REQUEST_METHOD'];
$output['SCRIPT_FILENAME'] = $_SERVER['SCRIPT_FILENAME'];
$input = file_get_contents('php://input');
if ($input)
  $data = json_decode($input,TRUE);
else
  $data = [];
$output = array_merge($output,$data);
header('Content-Type: application/json',TRUE,
       $status[$_SERVER['REQUEST_METHOD']]);
echo json_encode($output);
?>
```

   Make sure that there are no syntax errors in the file.

   b. In `$HOME/public_html/rw/.htaccess`, replace all occurrences of `file2.php` by `file3.php`, and save the file.

   c. Now, systematically send DELETE, GET, PATCH, and POST requests as you have done in Exercise 4c. The modified rewrite rule should redirect to `file3.php` instead of `file2.php` now.

   Observe what the response now looks like in </> RESTED or Postman and what the differences between DELETE, GET, PATCH, and POST requests are.

References

[1] Espen H. *</> RESTED Chrome Extension*. 23 July 2019. URL: https://chrome.google.com/webstore/detail/rested/eelcnbccaccipfolokglfhhmapdchbfg (accessed 27 November 2021).

[2] Espen H. *</> RESTED Firefox Extension*. 23 July 2019. URL: https://addons.mozilla.org/en-GB/firefox/addon/rested (accessed 27 November 2021).

[3] Postman, Inc. *Postman API Client*. 27 November 2021. URL: https://www.postman.com/product/api-client/ (accessed 27 November 2021).