Session 6: Web Services and HTTP

We have already had a look at writing web services and their clients. This week we will examine in more depth how modifying the HTTP request and response can help us develop maintainable and easy-to-understand web services. We will look at these topics:

- Returning HTTP codes from web services
- HTTP Authentication (advanced topic)

HTTP Methods: GET vs POST

HTTP - revision

Recall from Developing for the Internet that HTTP is a set of instructions which allow clients and servers to communicate with each other

- e.g. when you enter a URL in a web page, you construct an HTTP request for that page, and the server sends back a response

Recall also that HTTP requests and responses consist of ***two sections***:

- the ***header***, which describes the content
- the ***content*** itself, such as form POST data in the request, or the requested web page in the response

HTTP methods

HTTP comes with a set of standard request types, or ***methods***, to retrieve and manipulate URLs, which are specified in the HTTP request header. The two you've probably met are:

- GET - retrieve data from a URL. Query strings are GET requests.
- POST - send data to a URL (typically form data)

Normally, server side scripts (including web services) should use:

- ***GET*** for retrieving data (***no*** changes are made to the data on the server);
- ***POST*** for adding or updating data (changes to the data on the server ***are*** made)

So, for example, to add a new song to a database via a form, you should use a POST request while to search for all songs by a given artist, you would use a GET request. This would apply whether the server-side script is a web service or part of a regular website.

Why is this done? There are a number of reasons:

- ***Clarity***: correct use of GET and POST makes it obvious to people developing clients to your web service which scripts update the data and which merely retrieve data. Therefore, you help guard against accidental modification of your database by a developer calling a web service that they think just retrieves data (because it uses a GET request) but which actually modifies the database.
- ***Enhances security***: it is possible to set up a firewall to block POST requests while allowing GET requests. This is an easy way of restricting access to those scripts which update the database in some way.
- ***Restrictions on data which can be sent via GET***: browsers typically restrict the amount of data that can be sent to a server via a GET request. POST does not have these limits. Since add or update operations typically involve sending a relatively large amount of data to a server, it makes sense to use POST rather than GET.

The ***REST*** technique of web service development relies on correct use of appropriate HTTP methods.

Sending POST requests from clients

We saw how to use cURL to send POST requests last time.

Returning HTTP codes from web services

In Session 4 examined how the "download" web service could communicate its status, i.e. whether it returned successfully or not, to its clients. You probably used a simple status code like 0 or 1, or OK or BAD_ID, to indicate whether the web service completed successfully.

As you saw, returning a pure-data status code like the above means that any client, including a smartphone app, could connect to the web service and interpret the output to easily determine whether it completed successfully or not. HTML, by contrast, would only be understandable easily by a web browser. However, even this approach is, in some ways, not ideal as it is "reinventing the wheel". HTTP already provides a standard set of error codes to indicate success, or otherwise, to clients, so why not use that?

Recall that the first line of any HTTP response is a ***status code*** which indicates whether the request was successful or not. There are a large number of HTTP status codes including:

- ***200 OK*** - the page was retrieved successfully
- ***400 Bad Request*** - the request is in an invalid format
- ***401 Unauthorized*** - we try to access a page which we don't have rights to view
- ***404 Not Found*** - the page could not be found
- ***500 Internal Server Error*** - there was some sort of internal error on the server

Web services can use of these to communicate the success or otherwise of the operation. For example, a ***200 OK*** might be returned if the operation was successful, or a ***404 Not Found*** might be returned if we asked for a resource that did not exist.

Difference in usage of HTTP codes in web services versus normal usage

Imagine we had a URL to look up a given song:

```
http://hittastic.com/song.php?ID=1009
```

This script could return "404 Not Found" to the client if the song with that ID was not on the HitTastic! database, or, if an invalid ID (0 or less) was supplied, the script could return "400 Bad Request", another standard HTTP error code. Note that this use of error codes ***differs from the normal usage***:

- Normally, "404 Not Found" is used to indicate that a given file on the server does not exist. This error is generated by the web server software (e.g. Apache).
- Here, the file (***song.php***) ***does*** exist. It runs, looking up the song with the ID 1009. If no song with that ID exists, a "404 Not Found" is generated ***by the script*** to inform the client of the error.
- So, the error codes are being re-used by the web service to indicate an error generated by the ***script*** rather than the ***web server software***.

How to return HTTP codes from a PHP script

Simply use the ***header*** function with the appropriate HTTP code. For example:

```
header("HTTP/1.1 404 Not Found");
```

Remember to put the ***header()*** function ***before any output***. This is because it sets the status code line in the HTTP header and in an HTTP response, the header comes before the actual content.

How to test the HTTP status code from a client

If you are using cURL it's fairly straightforward. Once you have done your cURL request you query the ***CURLINFO_HTTP_CODE*** property to obtain the HTTP code returned by the web service, for example:

```
$connection = curl_init();
curl_setopt($connection, CURLOPT_URL, "http://remotese
curl_setopt($connection,CURLOPT_RETURNTRANSFER,1);
curl_setopt($connection,CURLOPT_HEADER,0);
```

```
$response = curl_exec($connection);
$httpCode = curl_getinfo($connection,CURLINFO_HTTP_CODE
echo "The script returned the HTTP status code: $httpCo
curl_close($connection);
```

Note how the line:

```
curl_setopt($connection,CURLOPT_HEADER, 0);
```

is still present. We can get the status code without having to include the full HTTP header in the output.

HTTP Authentication

You have almost certainly come across session-based authentication already, in which a session variable is used to determine whether the user is logged in or not. This method is all very well for simple websites but with web services it is less than ideal. The problem is that it ***introduces a dependency of one script on another***, meaning that a session-based login script must be run before a script which requires authentication (for example, a script to add a new song to the database). It also means that the client must keep track of the session: browsers do this automatically but some other type of client like a smartphone app might not. So in summary, ***a dependency on a previous login script is introduced***.

An alternative approach, favoured in web service development, is ***HTTP authentication***. In HTTP authentication, the username and password is sent ***within the HTTP header*** for every request to a script which requires authentication. This means that the script does not depend on a previous login script, or the existence of a session, because it receives the login details directly. There are potential disadvantages to HTTP authentication, for instance more requests containing the login details increasing the chance of interception, so if you are worried about "sniffing" attacks then you should set up a secure server.

Later in the unit we will examine the ***OAuth*** technique for authentication.

Sending the authentication details from a client

In cURL it is fairly straightforward to send the details from the client to the server. You simply set up the ***CURLOPT_USERPWD*** option and pass it the username and password separated by a colon. For example:

```
curl_setopt($connection,CURLOPT_USERPWD,"$username:$pas
```

will set the username and password to $username and $password, respectively. What is actually going on in the HTTP header though? The username and password are joined to form a string separated by a colon, i.e.:

```
username:password
```

and then ***base64 encoded*** (you have seen this already when looking at data URLs). The base64 encoded details are then placed in an ***Authorization*** line in the HTTP header. For example:

```
Authorization Basic VGltSm9uZXM6Y29sb3JhZG8xMjM=
```

where the sequence of letters and numbers is a base64-encoded username and password.

Reading HTTP authentication data server side

PHP makes it easy to read the login details sent by HTTP authentication. You make use of the ***$_SERVER["PHP_AUTH_USER"]*** and ***$_SERVER ["PHP_AUTH_PW"]*** variables. PHP automatically decodes the base64 encoded string, extracts the username and password from it, and places them in these two variables. These can then be compared against a database of users as normal. For example:

```
$user = $_SERVER["PHP_AUTH_USER"];
$pass = $_SERVER["PHP_AUTH_PW"];
// connect to database
$result=$conn->query("SELECT * FROM sc_users WHERE use
$row = $result->fetch();
if($row==false)
{
    header("HTTP/1.1 401 Unauthorized");
}
else
{
    // run the script
}
```

Note how we are handling errors in the login details. If no row in the database containing that username and password can be found, we send back a ***401 Unauthorized*** HTTP code to the client. The client can then read this code and react accordingly, for example display an error message and prompt the user to log in. Different clients can then react in different ways, for example a web client could display an error in HTML, whereas a smartphone app could display the error within the smartphone GUI and bring up a screen with a login form. As you should hopefully understand now, sending an HTML message would be inappropriate here as it makes assumptions as to what the client is: it may well not be a web browser!

Web Services and HTTP - Exercise

Standard Exercise

1. Modify your "download" web service (session 4) in the following way:
    - Communicate the "insufficient balance" error to clients using an HTTP code. Read the list of error codes on Mozilla to pick a suitable HTTP error code (between 400 and 499) to send back if the balance is too low. Any error code vaguely suitable should be picked. Do not be put off by statements like "not currently used" - if the HTTP code makes sense, use it!

   Upload it to your own space on edward2 and alter your client on the "fan" site (***clientdownload.php***) to talk to it. You will need to interpret the HTTP code returned.
2. Modify your download service further to send a 400 Bad Request if the ID is not supplied.

More advanced exercise

1. Add HTTP authentication to the "download" web service so that only authenticated users can download songs. A user must have a valid account in the "ht_users" table (present in the ***dftitutorials*** database on edward2; see http://edward2.solent.ac.uk/wad/users.php and PHPMyAdmin here) to be able to buy music. **This should replace the hard-coded user in the download script that you used before.**
2. Modify the "fan" site so that when a user downloads a song, they have to provide their login details before buying music through HitTastic! A login form, on a **new page**, should come up when the user clicks "Download". Pass the ID through to the login form as a hidden field. Then, POST the login data and ID to ***clientdownload.php*** and send your cURL request. The login details will be sent within the HTTP header (using HTTP authentication) from the "fan" site to HitTastic!