# INFO214 Assignment 3: World's Worst WebServer Report


**INFO214 GROUP**

Luke Hickin, Samuel O'Connel, Louie Murphy, Alister Butterfield

**SUPERVISOR**

Chris Edwards

# PURPOSE AND CONTEXT

The aim of this project was to create a RESTful web service in Python which supported GET and PUT requests.

The HTTP requests retrieve and store all of the websites information in an Oracle SQL Database, like that of a Content Management System.

Error handling and suitable response codes were also implemented for responding to user requests. Each request is also logged in the database each time a request is made.

No external libraries that help out with HTTP requests and responses were used in this project (only the socket library).

The purpose was to show an understanding of HTTP protocols, and web services which utilise REST in order to create a CMS like web server.

# SYNTAX & SEMANTICS OF COMMANDS

The implemented system currently stores data in the database in a REST software architecture style. Each 'resource' represents a URI, and each 'resource' is usually denoted by nouns, e.g. '/images/notes.png'.

Each resource can then be acted upon, i.e. one may wish to retrieve the data found in 'images/notes.png/', so the GET command will be used.

This web server currently supports these commands:

## GET

GET retrieves information in the database given a valid URI.

The client will request the information, specify the URI send other headers (which contain content type etc), and the server will send the resource back to the client .

An example GET will be in the format:

*GET /index.html HTTP/1.1*

*<new line>*

*<new line>*

This will retrieve the contents of index.html from the server.

### PUT

PUT  creates or updates information in the database given a valid URI.

The client will request that information be updated by sending a PUT request. As well as the request headers, the client will send the content they wish to create via the request body. The server extracts the headers and body, and will update/create the content in the database accordingly. The newly created or updated information is sent back to the client.

An example PUT will be in the format:

*PUT /index.html HTTP/1.1*
*<new line>*
*<new line>*
*[body]*
*<new line>*
*<new line>*

This will replace whatever is found at  '/index.html' with whatever was sent via the body.

NB. The URL '/' is unable to be modified in this web server, as it will always return a dynamically updated table of resources and links to for each one.

## ERRORS AND EXCEPTIONS

The server will currently respond to each request with at least one of these errors:

HTTP/1.1 200 OK  -  when a GET request is successful

HTTP/1.1 201 Created – when a PUT request is successful in creating

HTTP/1.1 400 Bad Request – when a request is empty/server does not understand

HTTP/1.1 404 Not Found – when the content asked for is not in the database

HTTP/1.1 500 Internal Server Error – when the server encountered an internal error (e.g. database problems)

## DESIGN DECISIONS

When developing the web server, code was constantly being refactored as lots of code was able to be re used multiple times. RESTful web services do require a lot of code reusability, so this was factored in when developing.

Error and response code handling was another major part of the design. It took a thorough understanding of when and where to apply each response code when designing the system. If the wrong reponse was sent to a client, the client may handle the response in such a way which will misinform the user of the outcome of the operation.

Python is an extremely powerful language when it comes to data manipulation. This made is much easier to manipulate the requests from the client when they were stored in strings. Headers could be extracted and searched through quickly, without involving much code. The http_handler method contains a lot of data manipulation in order to satisfy requests.

## ASSUMPTIONS AND LIMITATIONS

The current implementation assumes:

1. That any resource other than the table of resources can be manipulated
2. That different content types can be manipulated with information for another content type

Limitations/bugs that have been found in the implementation:

1. If the same resource is accessed multiple times within the same second, the timestamp will be the same when adding details to audit_log, therefore it will result in a violation of primary keys.