Instructions

Using Python, implement a web server that uses a simplified version of the HTTP protocol. Base your solution on the provided code templates. The use of external libraries and other built-in modules, packages or functions that are not already imported into server.py is not allowed. You should code your solution entirely in the server.py and submit it [to automatic grading system.](https://lem-grader.fri.uni-lj.si/grader)

Additionally, you are highly encouraged to write as many unit tests as you wish: the more the better. A starting point for unit tests has been provided in file tests.py. Although the unit tests that you write will not be graded, writing them will ease the development and help you in finding bugs. Note: Unit tests delete the contents of the database. I highly recommend using [TDD approach](https://en.wikipedia.org/wiki/Test-driven_development#Test-driven_development_cycle) where your first write the test (which fails), and then the functionality of your web server that makes the test pass.

Your solution will be checked with an anti-plagiarism software. All cheaters (those copying and those that provide material for copying) will be awarded with 0 points which means that they instantly fail this course.

Assignment specification

Your web server should support: \* Only HTTP requests following HTTP/1.1. This means that a valid request must contain a header with host field; \* Only two HTTP methods: GET and POST. When parsing POST requests, make sure the client is sending the content-length header field. It will tell you the number of bytes you have to read to parse the parameters in the body of the request. You can find a couple of request message examples [on this page.](https://www.tutorialspoint.com/http/http_requests.htm)

The server should be able to process both **static** and **dynamic** content. Static content denotes serving files from the file system (data inside folder www-data), while dynamic content denotes a simple web application that stores student data.

HTTP responses

While the HTTP protocol supports multiple [status response codes](https://www.tutorialspoint.com/http/http_status_codes.htm), your server should support only the ones mentioned below. To get a better idea of how HTTP response messages look like, visit [this page.](https://www.tutorialspoint.com/http/http_responses.htm)

200 OK

When the server receives a valid request, it should respond with a response code 200 OK. To prepare the correct response header, use the provided HEADER\_RESPONSE\_200 variable.

301 Moved Permanently

If you want to redirect the client to another page, use the 301 Moved Permanently response message. An example of a 301 message can be found [on Wikipedia.](https://en.wikipedia.org/wiki/HTTP_301) The important part is setting the response line and the appropriate headers, while the response body may be arbitrary (but it should still be meaningful to a human reader).

400 Bad request

If you receive a request which does not conform to the specifications, your should return a 400 Bad request message. As with 301, the important part is setting the response line and the appropriate headers, while the response body may be arbitrary (but it should still be meaningful to a human reader).

404 Not found

If you receive a request to a non-existent resource, the server should response it a 404 Not found message. Your response should be prepared with the help of the RESPONSE\_404 variable.

405 Method not allowed

If you receive a request using an invalid HTTP method, the server should respond with a 405 Method not allowed message. As with 301, the important part is setting the response line and the appropriate headers, while the response body may be arbitrary (but it should still be meaningful to a human reader).

Serving static pages

Serving static pages denotes serving a resource as-is from the www-data directory.

Methods

All static content should be served over requests using either GET or POST method. If another method is used, the server should response with a 405 message.

Response headers content-length and content-type

When responding, the server should appropriately set both the content-length and the content-type response headers. To find out the latter, you can use the guess\_type function from the mimetypes package to. If the function guess\_type returns None (i. e. cannot find out the file's mimetype), set it manually to application/octet-stream.

Parsing parameters

Always assume that the request parameters (be it either in the URL of a GET request or in the body of a POST request) are [URL encoded](https://www.tutorialspoint.com/http/http_url_encoding.htm). With POST requests, you can assume that the content-type is always set to application/x-www-form-urlencoded.

The string containing url-encoded parameters can be decoded using the unquote\_plus function from the urllib.parse package.

Serving directories and using the trailing slash

If the URL of the requested resource ends with a trailing slash (/, e.g. http://localhost[:port]/test/), the server should redirect the client to the index.html (using the same example, to http://localhost[:port]/test/index.html).

The client should be redirected by returning a 301 response status code, and the response header must contain the Location field. You can find an example of a such response message [on Wikipedia.](https://en.wikipedia.org/wiki/HTTP_301#Example)

If the URL of the requested resource does not have a trailing slash (nor does it have an extension), for instance http://localhost[:port]/test, you should be a bit more cautious. If there is a **file** called test, the server should serve it in the usual way. However, if there is a **folder** called test, the server should do the same as above: redirect the client to http://localhost[:port]/test/index.html.

If neither--the file nor the folder--exists, the server should respond with a 404. (You can check whether the folder exists by calling function isdir from the os.path module.)

Serving dynamic content

Implement a dynamic web application that allows users to store and retrieve student data.

Database (already implemented)

The web application requires a database. The database has already been implemented in a form of a file and two functions that read and write to it.

Use function save\_to\_db(first(str), last(str)) to add a user to the database, and use read\_from\_db(criteria(dict)) to retrieve them from it. (Again, do not forget: running unit tests will delete the contents of the database.)

For instance, calling save\_to\_db("Janez", "Novak") will create a new DB entry with the first name Janez and the last name Novak. Additionally, the database will generate a unique number for this entry.

Each entry is internally represented as a Python dictionary with three string keys: number, first and last. Key number represents a unique identification number, while keys first and last represent the first and last name. Here’s an example.

student = {"number": 1, "first": "Janez", "last": "Novak"}

Function read\_from\_db(criteria(dict)) will return a list of such dictionaries. You can call the function with an optional argument of type dict to further limit the results. For instance, the following invocation will return all entries where the first name is set to Janez and last name is set to Novak.

students = read\_from\_db({"first": "Janez", "last": "Novak"})

*# students is a list of dicts*

As criteria, you may provide an arbitrary combination of keys using fields number, first and last.

Design (already implemented)

The design (HTML and CSS) is implemented within files app\_list.html, app\_add.html and user\_style.css. Do not modify these files.

Dynamic application

The application should respond to the following two (virtual) URLs. (We call these URLs virtual, because the resources (app-add, app-index, and app-json) do not exist in the file system, although to the client it might seem as they do.)

Adding entries to the database

URL: **http://localhost[:port]/app-add**

On this URL, the web server should only accept POST requests. Each request has to contain two parameters: first, containing the student's first name, and last, containing the student's last name.

If the request contains both parameters, add an entry to the DB, return code 200, in the body of the HTTP response, return the contents of the app\_add.html.

If any of the parameters is missing, the server should respond with a 400 error message. If the request method is not POST, the server should respond with 405.

Reading and filtering data in HTML

URL **http://localhost[:port]/app-index**

On this URL, the server should only accept GET requests; all other request types should be responded with an appropriate error code. If a request contains no parameters, simply read all entries from the database, and prepare your response with code 200.

Prepare the body of the response using the contents of app\_list.html. Before sending the response body to the user, make sure you replace the {{STUDENTS}} placeholder with the actual information about students. Use the ROW\_TEMPLATE variable to display data for every student -- every student should be represented with a row in the table.

TABLE\_ROW = """

<tr>

<td>%d</td>

<td>%s</td>

<td>%s</td>

</tr>

"""

The ROW\_TEMPLATE variable contains three placeholders for student data: replace the %d with student’s number, the first %s with student’s first name, and second %s with students last name.

Additionally, the GET request might contain criteria parameters for limiting the amount of students that will be displayed. The criteria could contain three parameters: number, first and last. If any of these request parameters is set, use it to filter the list of students. For instance, a GET request to http://localhost[:port]/app-index?first=Janez should return a list of students whose first name is Janez.

Reading and filtering data in JSON

URL **http://localhost[:port]/app-json**

On this URL, the server should only accept GET requests; all other request types should be responded with an appropriate error code. If a request contains no parameters, simply read all entries from the database, and prepare your response with code 200.

The response should be rendered as a message in JSON. You may use the Python's built-it module json and its function json.dumps(object); the data from the database can be passed into the dumps function directly: json.dumps(read\_from\_db()).

The result must be returned in the response body. When returning JSON messages, one must set the content-type to application/json. An example response is given below.

HTTP/1.1 200 OK

content-type: application/json

content-length: 256

connection: Close

[

{

"number": 1,

"first": "Janez",

"last": "Novak"

},

{

"number": 2,

"first": "Marija",

"last": "Novak"

},

{

"number": 3,

"first": "Cirila",

"last": "Novak"

}

]

Additional requirements

Remember, you are not allowed to use any functions, modules or packages aside from those the server.py has already imported (mimetypes, pickle, socket, os.path.isdir, urllib.parse.unquote\_plus, and json) or those that you write yourself.

When writing unit and integration tests, there is no such constraint: some tests already a third-party library called [Requests](http://docs.python-requests.org/) for sending HTTP requests and parsing HTTP responses.

The grading will be done automatically using integration tests, similar to those in file tests.py: every required functionality will be accompanied with an integration test that will start your server, send an HTTP request to it, and then validate server's HTTP response.

During grading, each test will start your server by invoking function main(port(int)). So make sure that your server is listening on the appropriate port number and not on some hard-coded value (like 8080).

Feel free to either pose your questions in the forms or ask in person during office hours or lab sessions.