**IMPORTANT:**

Do not start programming directly with this assignment! Make sure that you understand the problem and what is requested of you. Make models and divide the problem *before* writing a single line of code! This assignment will be challenging, and a lot of extra time will be needed if an ad-hoc solution is attempted.

**Introduction**

A client-server application can be simplified to describe a model wherein a program (the server) receives and handles requests of services done by another program (the client). The server program awaits requests done by the client program and begins working on a request as soon as it is received. The two programs are often referred to as one application.

The client-server model is used all around us; websites, chat programs, and email are just a few examples.

**The Server**

For this assignment, the file-managing server is required to handle **multiple** clients connecting to it where each client might work in a directory of their own and with files inaccessible to other users. If a user is already logged in, further requests to log in with that username should be denied. However, the server is only required to be able to handle one service request at a time. Do note that service requests done by different users should **not** interfere with one another unless necessary. For instance, if two users use the command "read\_file" but with different files, then subsequent calls done by these users are to handle the specific user's file and not any other file.

The server must be able to handle two different kinds of users—**user** and **admin**. Some service requests must be denied if done by a **user** and not an **admin**.

**File Structure**

A sample directory structure is provided with this assignment but is not required for use. Note that this structure is created with two users in mind, one **admin** and one **user**. Service requests are never to be allowed to traverse outside of a given server structure. The structure can be found [here](https://bth.instructure.com/courses/1788/files/175104/download?wrap=1).

Root  
|---Admin  
       |---top\_secret\_data  
              |---pictures\_of\_cats     
       |---not\_so\_secret\_data  
|---User  
       |---pictures\_of\_dogs  
       |---py\_code

Registered users, folders, files, and other implementation data **must be saved between sessions!** Users created during one session must be usable the next time the server is started, and all folders, files, and data are to be properly accessible.

How the server keeps tab on this structure is up to you; it is strongly recommended that you start this assignment with sitting down in your group and define how the server keeps track of all files.

**Services**

The server is to be supporting a subset of commands commonly found on *UNIX-*based systems. These commands are heavily simplified compared to their *UNIX* counterparts and are the services of the server. Please refer to the table below for the different services that are expected of the server, as well as how they are to be used. If a request of service is conducted with an unknown command or an erroneous input, proper information is to be passed back to the client.

|  |  |  |
| --- | --- | --- |
| Service command | Description | Options |
| change\_folder <name> | Move the current working directory *for the current user*to the specified folder residing in the current folder. If the <name> does not point to a folder in the current working directory, the request is to be denied with a proper message highlighting the error for the user. | To walk back the previous folder, a <name> of two dots (..) is provided. This should *only* be available if the user is not currently standing in Root. This overrides the <name> rule in the previous box. |
| list | Print all files and folders in the current working directory for the user issuing the request. This command is expected to give information about the name, size, date and time of creation, in an easy-to-read manner. Shall *not* print information regarding content in sub-directories. | - |
| read\_file <name> | Read data from the file <name> in the current working directory for the user issuing the request and return the first hundred characters in it. Each subsequent call by the **same** client is to return the next hundred characters in the file, up until all characters are read. If a file with the specified <name> does not exist in the current working directory for the user, the request is to be denied with a proper message highlighting the error for the user. | A service request without a <name> variable should close the currently opened file from reading. Subsequent calls with this file as <name> should start reading from the beginning of the file. |
| write\_file <name> <input> | Write the data in <input> to the **end** of the file <name> in the current working directory for the user issuing the request, starting on a new line.  If no file exists with the given <name>, a new file is to be created in the current working directory for the user. | If <input> is empty, then the specified file is to be cleared of content. |
| create\_folder <name> | Create a new folder with the specified <name> in the current working directory for the user issuing the request. If a folder with the given name already exists, the request is to be denied with a proper message highlighting the error for the user. | - |
| register <username> <password> <privileges> | Register a new user with the <privileges> to the server using the <username> and <password> provided. If a user is already registered with the provided <username>, the request is to be denied with a proper message highlighting the error for the user. A new personal folder named <username> should be created on the server. The <privileges> should be either user or admin. | - |
| login <username> <password> | Log in the user conforming with <username> onto the server if the <password> provided matches the password used while registering. If the <password> does not match or if the <username> does not exist, an error message should be returned to the request for the client to present to the user. | - |
| delete <username> <password> | Delete the user conforming with <username> from the server. This service is **only** available to users with a privilege level of **admin**, and <password> is to be the password of the **admin** currently logged in. If the request is done by a user that does not have **admin**privileges, <password> does not match or if the <username> does not exist, an error message should be returned to the request for the client to present to the user. | - |

The server is to be assumed listening for connections on localhost (127.0.0.1), port 8080.

**The Client**

For this assignment, the client is taking a more passive role. When started up, it should present the user with the option to either login or register to the server.  These options are to result in requests to the server, and a login is only to be allowed if the username and password matches with the information on the server.  Once logged in, the client should have the user's personal folder as starting point to work with.

The user should be able to input any command described for the server. Whenever a command is issued, the client is to send a request to the server and present the result to the user if applicable. The client is also required to save down all commands issued by each user.

Lastly, the client got two commands that they are to directly handle themselves:

|  |  |  |
| --- | --- | --- |
| Command | Description | Option |
| commands | Print information about all available commands, including expected input and what alternative usage they have. | If this command is followed by the string "issued", then all commands the user has sent, including input, are to be presented on the screen in an easy to read fashion.   If this command is followed by the string "clear", then all commands issued by the user is to be cleared from the client. |
| quit | Logout the user, close the connection to the server, and close the application. | - |

**Testing and Error Handling**

You are expected to write tests using asserts as well as either the doctest or unittest module for both the server and the client. It is up to you to decide exactly how many tests that are to be written.

The tests implemented with asserts are to be used for simpler tests, for instance the input to functions.

The tests implemented with the doctest or unittest module are to be clearly documented in a separate report. This text must highlight **why** these tests where chosen, **what** they test, and **how** it is tested. This report should not contain code, but be an overview of the reasoning behind the test and how these tests were identified as necessary.  The report is also required to contain a section arguing why the number of implemented tests were deemed enough and what areas of the code that are not tested.

**Report**

A report are to be written outlining how work on this assignment was conducted. The report should clearly specify the work done before coding started, how the work has been divided between members of the group. It should also describe different problems that emerged while working and how these were handled.

**Group Requirements**

You are free to group yourself together with up to two other students, resulting in groups of 1 to 3 students. Once a group is decided, you must:

1. Create a ***private*** github repository for this assignment,  shared between the members in the group, as well as with [cnl@bth.se,](mailto:cnl@bth.se,)[ajx@bth.se,](mailto:ajx@bth.se,)and [sivadasari.bth@gmail.com](mailto:sivadasari.bth@gmail.com).
   * Free private repositories can be created with a github account created through your school email, see [https://education.github.com/students (Links to an external site.)](https://education.github.com/students)
2. Send one email per group to Carina Nilsson containing a chosen name for the group, and each group members':
   * Name
   * Social security number

The last date to apply to work with this assignment in a group is *October 13, 2019*. After this date, all students that are not in a group will be placed in a single member group. Note that single-member groups should also have a ***private***github repository for the assignment. This repository must be shared with the teachers.

You are expected to share the workload evenly in the group. Submission history, with a focus on content and quality, not number of lines, will be part of the **individual** assessment.

**Code Requirements**

* A pylint value below 8/10 will result in a failed grade
* Your code must be logically divided with clear connections.
* Proper docstrings are expected for all classes, functions, and modules.
* The logic for the server and client must be divided into different files. Everything cannot be implemented in one file.
* All code should be written as clearly and consistently as possible.
* Inheritance must be identified and implemented where relevant.
* Only modules available on the computers in H320, H321, and H322 are allowed to use for solving this assignment. A list of the modules can be found [here.Preview the document](https://bth.instructure.com/courses/1788/files/183391/download?wrap=1)

Submission

As a group, submit the following to Canvas:

* A zip-file containing your *entire* implementation, including directory-structure for the server
* Clear instructions on how to run your implementation
* The report regarding the work as a pdf-file
* The report regarding the tests (pdf, can be included in the other report document)

**Remember:**  
It is not allowed to submit code written by anyone but your group members. Your submission will be checked for plagiarism. Suspected cases of plagiarism can be brought to the disciplinary board, and in worst case lead to suspension from your studies.