Outlab 6 : Git & WebStack

Please refer to the general instructions and submission guidelines at the end of this document before submitting.

P1. Up and running [65 points]

One of the more popular system architectures is the client server model, wherein one machine acts as a server and other(s) as client(s). The responsibility of server is to reply to requests generated by client(s). A request generally consists of a data to locate the server and some questions that a client has. The data regarding location of the server is used by Networking elements (like routers, switches etc..) to guide the request to the server. Once a request reaches the server, the questions asked by client are processed and answered back in a response. Response (similar to request) contains location of the client and the answers to the questions.

In server client architecture, one of the more popular & classic request-response model is the webpage model. In webpage model a request is generally a string that contains both the location of server and the questions (generally in terms of parameters). And a response is a web page. This is where HTML (Hypertext markup language) comes into picture. You see HTML is used to create web pages. So a webpage response is actually HTML script. HTML is a markup language. HTML is not a programming language. There is no concept of variables/data containers/if else/loops in HTML.

HTML (for that matter most of WebStack) is highly autodidactic. This is one of the better references for the entire webstack which has inbuilt tutorials.

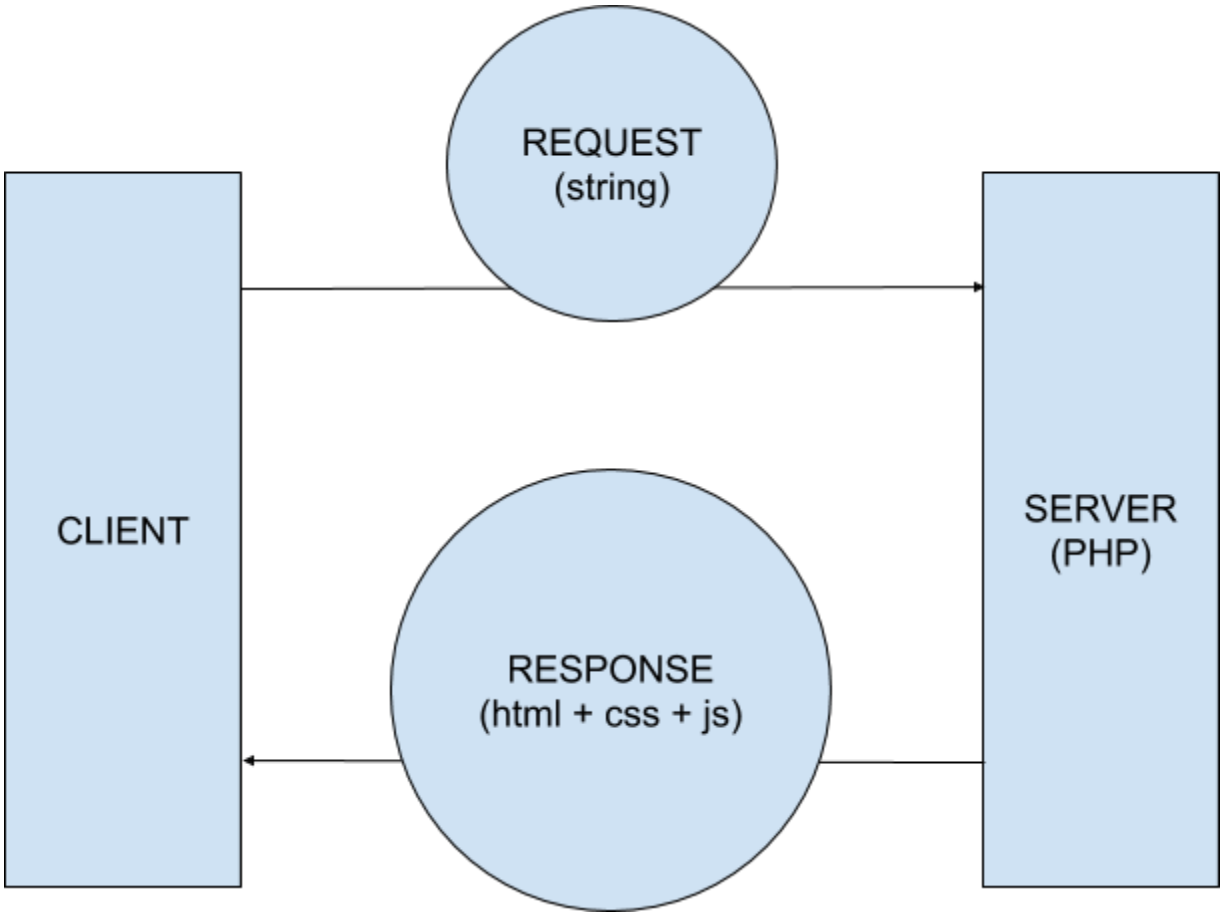
HTML itself is sufficient for representing data. But over time a need for sophistically organizing and visualizing data was felt and one of the products of that is CSS (Cascading Style sheets). As the name suggests a CSS script styles a page generated by HTML script. This is an example of a webpage where almost no CSS is used. This is an example of a webpage where CSS is used properly. This is an example of web page where CSS is not used properly.

HTML and CSS by themselves are enough to generate and organize data. But a webpage created solely by HTML and CSS is static (just a bunch of strings). To bring life into a webpage JS (Javascript was introduced, not to be confused with Java). JS is a full on turing complete

programming language. It can do stuff. Mainly JS is used for manipulating HTML and CSS content in the context of webstack. Thus it brings a sense of dynamism to a webpage.

Note that all of the above discussed scripts HTML CSS and JS are all part of response. When the response reaches client, client (generally a browser) interprets it and displays what you daily see as web pages.

The following diagram summarizes what all is discussed until now



PHP (Hypertext Preprocessor) (also a full on turing complete programming language) is tailor made for creating responses to client requests. Typically a server side (backend developer) writes a PHP script which accepts parameters (questions) from client and outputs an appropriate response contains HTML CSS & JS scripts.

Task 1

In this task you are going to build your homepage. This is going to stay with you for the rest of your IITB life. If people want to know something about you the first thing they do is to checkout your homepage (um... maybe facebook first, but your homepage shall have extensive details).

1. Your homepage should be a reflection of you. So it should typically includes a short brief about yourself, your interests, your projects, your ambitions and goals, your public contact information
2. To start off first create a webpage that is accessible using the url (*www.cse.iitb.ac.in/~<cseuserid>/,* referred to as **root url** from here on).
3. This page should contain
    a. Your name.
    b. **A one line description about yourself**, who are you as a person?
    c. When clicked on your one line description a **modal** should popup, which shows your biography.
    d. A **carousel** of your public pictures.
    e. A page hit counter which shows number of times the page was accessed. This is a measure of popularity of your page. It should (obviously) increase by 1 for every access. This can be done by storing a counter in a separate file, reading from file, sending the counter in web page, incrementing it by 1 and storing it back.
4. Make a page showing **your interests, ambitions and goals, your hobbies, your favorites** etc…
    a. You can use **dropdown/tabs** to neatly represent each section
5. Make one page for your **projects** (done, ongoing, future planned) etc…
6. Make one page for your **CS251 team**
    a. team members, and their profile pictures
    b. team name and why you choose that name
    c. a logo of your team
    d. a real-time clock showing current data and time refreshing every second
7. Make a page which shows your **contact information**. This will be public so be careful.
8. Make a page about your academics.
    a. Use CSS **accordions** and **collections** to group the courses you took under the semesters you did them.
    b. Create your current timetable using **table** tag
9. Make a page which contains a **form** containing fields **Name** and **Comment and** a **submit button**.
    a. Anyone who comes to this page can add a comment about your homepage and click submit.

    b. Submission should not occur if any of the fields are empty.
    c. All such comments must be saved in a file.
10. Make a page which reads all the comments in the file in the previous task and displays them in a neat **collection**.
11. A page/webpage in this task may refer to a html or php file based on context and use.
12. Give titles to each page appropriately. Give a common icon (which is displayed to the left of title. Icon should not be inside body. For example google.com has a little G icon) for all pages.
13. All pages should contain (the same) **menu bar** which contains links to all web pages you are going to create. You can give any names for your pages/files.While grading we will first go to your root url and use menu bar to navigate between pages. So make sure you connect all web pages in menu bar and keep your website up.
14. Style everything neatly. Bootstrap and Materialize CSS are good frameworks for that.
15. As you can see this task is largely subjective. The marks will be given on aspects including but not limited to
    a. Content
    b. Functionality
    c. Aesthetics
    d. Interesting features
16. Put working homepage urls of three members one in each line in a **README** file so that we can directly open them. This file should also contain % contribution by each member (each scaled to 100% i.e. if only two people did all the work, contribution should be 100% and 100% for those two and other one should have 0% contribution). This way if someone did no work at all the least functioning/good looking one out of other homepages will be taken into account for grading.
17. All three members should submit a copy of their working homepages. Do not change online version after submission. We shall randomly check if the submitted version of homepage matches with online version. If a mismatch occurs there will be severe consequences. Checkout submission instructions.

Task 2

1. Write **true** or **false** (not True, TRUE, true. etc...) for each of the following one question per line in that order in **trueorfalse.txt**
    a. PHP is dynamically typed language.
    b. Javascript is statically typed language.
    c. PHP has duck typing
    d. JS does not have duck typing.
    e. PHP is a compiled language.

f. JS is a compiled language.
g. PHP does not have types.
h. JS does not have types.
i. HTML is a compiled language
j. A php file can contain segments of C++ code in it.

Task 3

Go to your facebook account and open one of your post. Use inspection tool in your browser to change the number of likes to 100K. Take a screenshot of the modified post and name it **fb.png.** Does this mean the #likes are changed permanently?

P2. Git it? [35 points]

Git offers various features to control and make easy collaboration across users, provided you know what features could be used for what kind of collaboration.

It is usual practice to have the implementation of basic APIs and code with final updates on the master branch, while having implementations on separate branches, evaluated, finalised and only then merged into the master branch. We shall attempt doing this for a basic sorting library built in C++.

Submission:
Add the user **varshiths1** to the repository on git.cse.iitb.ac.in with developer permissions. Name the repository **<rollno1>-<rollno2>-<rollno3>-git**. If the name is not available, use extra digits at the end.

Note:
The person with the least (lexicographically smallest) roll number is referred to as the *first user*. The others as *second user* and *third user* accordingly. The commits made by the users are to be made on separate systems, with the only way of sharing code being through the remote repository.

Task 1: Basic API

1. Create a new repository git.cse.iitb.ac.in on the account of the *first user*. Add all the other users as collaborators.

The following actions are to be performed by the first user on the master branch on his/her local clone.

2. Clone the repository.
3. Create a file **sorting.h** that declares the following function:
   vector<int> sort_custom(vector<int>);
   The function is to takes as input a *vector<int>* and return a *vector<int>* with the elements in the ascending order.
4. Create a file **main.cpp** that includes this header, reads in *n* elements from *stdin*, initialises a *vector<int>*, calls the function *sort_custom* on the vector and prints out the *n* elements onto *stdout*.

Usage:
$ ./main < inp > out
inp:
5
1 6 2 7 10
out:
1 2 6 7 10

5. Make a commit with the message "add API and data read".
6. Create a file **sorting.cpp** with the implementation of the function declared in the header using a simple bubble sort algorithm.
   Make a commit with the message "add basic implementation".
7. Add a **Makefile** to compile the files and create an executable **main** upon running *make*.
   Make a commit with the message "add makefile".
   Make sure the compilation and implementation are correctly working.
8. Push your commits onto the remote repository onto the master branch.

Task 2: Branching

Now that the first user has the base repository ready, he/she decide to invite other collaborators to improve upon his/her code base.

The second and the third users now wish to improve upon the implementation of the sort_custom function. They want to be able to work simultaneously without cluttering each others' work with commits. They work on different branches.

The following actions are to be performed on the second and third users' local clones.

1. Clone the repository or pull updates if already cloned.
2. Create a branch named *merge-sort* and checkout to it (check it out?).

3. Change the implementation of sort_custom from bubble sort to merge sort. Make a commit with the message "*change implementation to merge-sort*".
4. Push your commit(s) to a remote branch *merge-sort* (notice that the name of the remote branch could be different from the source branch).

The third user independently performs the above steps for insertion sort, in his/her own repository. The names of the corresponding branches are to be *insertion-sort* and the message, "*change implementation to insertion-sort*".

Task 3: Merging

Now the first user wishes to update the master branch with the most efficient implementation (which is (arguably) merge sort). The third user's implementation is ignored (like it does in many cases).

The following actions are to be performed by the first user on his/her local repository.

1. Pull the changes from the remote branch *merge-sort* into the master branch. There might be a merge conflict since the changed implementation overwrites the previous algorithm. In other cases, since there are no commits on master after the branch originated, the pull might be performed in something called a "fast-forward mode".
   Either;
   1. Override this mode to ensure that a merge commit is created.
   2. Don't override the fast forward mode, but make an additional commit with dummy changes (maybe add a comment) to make a note that the code base has been checked after the merge.

   The commit should have the message "update implementation to merge sort". Push the updates.

Task 4: Updating the API

The first user, being the dictator of the API of the library decides to change the specification of the function to include sorting a sub array.

The following actions are to be performed by the first user in his/her local repository on the master branch.

1. Change the function in **header.h** to the following.
   vector<int> sort_custom(vector<int>, int startidx, int endidx);

The function is to takes as input a *vector<int>* and return a *vector<int>* with the elements in the sub array [startidx, endidx) in the ascending order and the rest of the elements in the same location. *Assume 0 <= startidx, endidx < vector_length*
Make a commit with the message "update API to add subarray sort". Do not make any changes to sorting.cpp. Push the changes to the remote branch master.

The first user asks the second user to update his/her implementation to reflect the API change. The second user now has two options:

- Merge the API change from the master branch into the merge-sort branch, and then make changes to the function and commit.
- Change the entire branch as though the branch originated, not from the commit with the message "add makefile" (as before) but from the latest commit itself (commit with message "update API to add sub array sort").

To execute the second option, the second user has to perform a ***rebase***. This applies the changes in the commits originally in the branch to the new "base" and in the process, creates new commits to replace the old ones.

The following actions are to be performed by the second user on his/her local repository.

2. Update all branches.
3. Checkout to merge-sort branch.
4. Perform a rebase to shift the base of the current branch to the latest commit on the master branch.
5. The resolve the conflicts that (if they) occur upon rebase, choosing the newer API.

   Note: In this particular scenario, the API change is not significant, so the utility of rebase may not be obvious. But if there is a sub-routine that is being updated on master, the rebase helps in each commit reflecting the updated implementation of the sub-routine, which could be helpful for developers who want to branch out from such a commit.

6. Make the updates to the implementation to the function. Make a commit with the message "update merge-sort to reflect API update".
   Try to push the changes to the remote *merge-sort* branch. It fails. Try to understand why. Override using git push --force.

The following action is to be performed by the first user on his/her local repository.

7. Pull the updates from the remote branch merge-sort into the master branch, merge the changes to create a commit *(A)* with the message “update implementation to reflect API update”.
   Either perform the pull in “fast forward mode” and add dummy changes to make the commit *(A)*, or override the fast forward mode while making the pull. This creates the commit *(A)*.

8. Make changes in the usage of the function in main.cpp entering the correct arguments to perform the same task. Commit your changes with the message “update usage in main.cpp”. Push the changes.

General Instructions

- Make sure you know what you write, you might be asked to explain your code at a later point in time
- Grading may be done automatically, so please make sure you stick to naming conventions
- The deadline for this lab is **Sunday, 26th August, 23:55.**

Submission Instructions

After creating your directory, package it into a tarball **`<rollno1>-<rollno2>-<rollno3>-outlab6.tar.gz`** in ascending order. Submit once only per team from the moodle account of smallest roll number.
The directory structure should be as follows (nothing more nothing less)

```
<rollno1>-<rollno2>-<rollno3>
      ├── <cseuserid1>
      │   └── files ...
      ├── <cseuserid2>
      │   └── files ...
      ├── <cseuserid3>
      │   └── files ...
      ├── fb.png
      ├── README
      └── trueorfalse.txt
```