**Week 2**

**2.1 Revision of Promises and Async JS**

* JS is single-threaded. The async task is handled by the C++ code of the browser in the case of setTimeout whereas the OS handles the task in the case of reading from a file.
* ENOENT. Status code for file not present.
* Callback hell. When there are async functions inside async functions.
* Promises help us write our own async functions.
* Non-promisified function does not return anything and takes callback as an input whereas promisified function return a promise and does not take callback as an input.
* Cases where we would need to do asynchronous calls: network call, sleep/wait for some time, read a file, database call.
* As there is .then that we can call on a promise, there is also .catch which catches errors if the promise if not returned/the file is not read etc.
* Promise Chaining. Key to get rid of callback hell. See code. It is slightly faster than Promise.all().
* We can also have reject besides resolve in Promise(function(resolve, reject) and then we can use .catch after .then.

**2.2 NodeJS Runtime and HTTP**

* NodeJS is a runtime. It is not a language. Runtime is basically just a bunch of C code or Zig code that compiles JS to machine language 0s and 1s.
* Burn is written in Zig which is much faster than NodeJS.
* NodeJS is used for creating HTTP servers.
* HTTP protocol is used for website’s frontend to talk to its backend.
* HTTP server is some code that follows the HTTP protocol.
* Cookie is one of the many headers that we can send. It lets the backend server know that the user is logged in. It is called Authorization in Chrome’s Dev Tools. It stored the login hash.
* Client needs to send: Protocol, URL, Route, Header, Body, Method.
* In address bar, we have protocol, url, port, and route. For secure sites, port is 443 which is usually not shown.
* GET and POST are some methods.
* Server needs to reply back with: Response Header, Response Body, Status Code.
* Domain Name Server resolution. Google.com has a server. Instead of writing the server’s IP address, we write google.com so that the DNS maps to google.com’s IP address.
* When you retrieve information, you send a GET request. When you add some information like starting a conversation with ChatGPT, you send a POST request. When you update information, you send a PUT request. When you delete information, you send a DELETE request.
* It is not mandatory to follow these practices, but it is good practice.
* NodeJS is an external library like fs which is used to read/write files.
* NextJS is becoming more popular backend servers.
* HTTP server can get blocked in Java, Golang.
* When we go to localhost:3000 that is when the code starts running.
* Expensive calls like database calls are asynchronous.
* If we have two HTTP servers running, we can run them on different ports.
* REST APIs are similar to HTTP servers, but they can only be accessed by other NodeJS processes.
* The backend can also return HTML.
* Always make sure to restart the process.
* Extra slash at the end cancels out.
* You can access your local server on different devices but on the same WIFI using your IP address. Write actual\_ip\_address:3000 in your other device’s browser.
* Server can host multiple applications and multiple domains can point to the same server. Whenever a request comes, the server can check from which URL the request came from.
* We can’t access body using express, so we use body-parser.
* NextJS writes you both front-end and back-end.
* HTTPS, S takes you automatically to port 443.
* When the client side performs a post operation, we write app.post in code.
* Req and res are given by app.post.
* We can write `export PORT=3000` on terminal and set `const port = process.env.PORT` in code. The same purpose of writing `const port = 3000` is achieved.
* You write which server to hit in the HTML.
* If we don’t write app.listen, it immediately exits. It’s similar to writing a function but never calling it. App.listen actually starts the server.

**2.3 Bash and Terminal**

* pwd 🡪 prints working directory.
* cd ../.. 🡪 takes you two folders back.
* touch 🡪 for creating file.
* cat filename 🡪 seeing the content of file.
* vi filename 🡪 for editing files on terminal. Complicated. Esc + : + q + ! to exit without saving. Esc + : + wq + ! to save and exit. ! mark is optional.
* mv filename foldername 🡪 for moving file to a specific folder.
* mv foldername1 foldername2 🡪 moving foldername1 to foldername2.
* cp filename foldername 🡪 copy files. Similar to move but copies instead of moves.
* cp -r foldername1 foldername2. Directories cannot be directly copied, we have to give -r for recursive.
* nvm 🡪 node version manager. For installing node.
* npm 🡪 node package manage.
* Express is an external dependency which does not exists locally but on the npm registry.
* npm install libraryname 🡪 for installing external library or dependency.

**2.4 Advanced Bash**

* ls -l 🡪 for seeing more information about your files.
* ls -R directoryname 🡪 for seeing information about each individual file in the subdirectory. R is for recursive.
* ls -t 🡪 for displaying files in the directory by the order they were last modified. You can also combine these like ls -lt.
* ls -la 🡪 for seeing hidden files like .DS\_Store.
* ls -lr 🡪 for listing files in reverse order of modified by.
* ls – s 🡪 for listing directories by size.
* ls -lR | grep .json 🡪 recursively grab all the json files from this folder.
* ls \*.js 🡪 \* is a wildcard. You can also use wildcard instead of grep.
* ls Zoo\* 🡪 gives all files which have Zoo in their name. It also gets those files which have Zoo in the middle.
* ls .. 🡪 listing content in parent directory.
* cat > filename 🡪 for editing the file. Ctrl + D to save and exit.
* cat >> filename 🡪 appending more data to an existing file.
* mkdir test && cd test 🡪 create directory and go to that directory.
* mkdir -p frontend/scripts 🡪 lets you create directories recursively. If we run another command after this like mkdir -p frontend/css, it will create a new subdirectory in frontend. Very clever.
* mv currentfilename newfilename 🡪 renaming file name.
* mv currentfilename path/to/directory/newfilename 🡪 moving the file and renaming.
* rm filename 🡪 for deleting file.
* rm -r foldername 🡪 for deleting folder.
* chmod u+x filename 🡪 chmod + ‘ ‘ + u/go (user or group) + (+/-)(add/delete) + r/w/x + ‘ ‘ + filename 🡪 changing permissions.
* For folders we would have to use -r.
* If we only want to do it with numbers. 4 is for read, 2 is for write, 1 is for execute.
* If we have -rw-r--r-- as permissions, then to give write permission to group we would give chmod 664 filename. This will turn permissions to -rw-rw-r--.
* echo “Hello World!” 🡪 displaying hello world.
* Echo $PATH 🡪 displaying path.
* head filename 🡪 similar to cat, but gives first ten rows of the file.
* tail filename 🡪 similar to head.
* head -20 filename
* command1 | command2 🡪 pipe character. Whatever output comes from command1 flows to command2.
* tail -n +25 filename | head -5 🡪 viewing lines 25-30.
* wc filename 🡪 viewing number of lines, word count, character count.
* grep helps you to find out occurrences of a specific word/regex/phrases in a folder/file etc.
* grep “one” newfile.text | wc,
* grep -c “one” newfile.txt 🡪 count of the number of lines in which this particular occurrence happened.
* grep -h “one” newfile.txt 🡪 this will give the matched lines.
* grep -hi “one” newfile.txt 🡪 ignores the case.
* grep -hir “one” . 🡪 gives all the occurrences in the folder recursively.
* grep -hin “one” newfile.txt 🡪 also gives the line numbers now.
* grep -o “one” newfile.txt 🡪 only picks “one” which are separate words and not in the middle of other words.
* grep -o “one” newfile.txt 🡪 not sure, but this is the default.
* history 0 🡪 for viewing history.
* We can also create .sh files which run bash commands. These files start with `#!/bin/bash` and then have the usual bash commands that we write on terminal. To run these files, we write bash filename.sh in terminal.
* grep -v “INFO” log.txt 🡪 omit all the logs that have INFO. Gets all the details, apart from INFO.
* grep -A 5 ERROR log.txt 🡪 see 5 lines after error.
* grep -B 5 ERROR log.txt 🡪 see 5 lines before error.
* grep -C 5 ERROR log.txt 🡪 see 5 lines before and after error.
* sed is used for finding and changing content in the file. Finding is similar to grep. sed has all the functionality of grep.
* awk is also similar. awk has all the functionality of sed.
* Look into sed and awk later. Too advanced. Really good for log files, changing errors, counting errors, etc.

**2.5 Express with Examples**

* HTTP servers lets you expose your logic over the internet.
* If n is a query parameter, we pass it in the address bar using ? + n = 30. If we want to add more parameters, we use & and write the other parameter.
* Whenever we’re sending back, we send back a string so that it does not think that it is a status code.
* Express is not directly bundled with node. It does not come with node.
* Until we deploy it on an AWS server, it would not be accessible by the internet. But we have deployed it on our local network.
* Anything after the question mark, is not counted in the route.
* To catch the query parameter, we use req.query.n.
* If we don’t specify the query parameter in the address bar, it will take 0 as default.
* Req and res stands for request and response. Anything related to input, headers, parameters is related to request. Anything related to what data to send back, what status code to send back is handled by response.
* If the response is in the series of 200, then it is fine.
* When the route does not exist, the server usually responds back with 404.
* If there’s an error, the server usually responds back with 500.
* If there are wrong inputs, then it’s usually 411.
* If you’re banned, then it’s 403.
* See example in code.
* For get requests, the popular input type is query parameters.
* For post requests, you send data in the body.
* If we just go to a browser, we send get requests.
* For now, we are using Postman to send post requests.
* /file/:filename so that whenever someone types /file/1 or /file/a.txt, the control reaches the same get request.

**2.6 Filter, Map, and Arrow Functions**

* Arrow functions is just another way to write normal functions. But there’s a difference of how they get bonded.
* Map expects a function which transforms the array.
* You can use arrow functions in map and filter.
* See examples of arrow functions, map, and filter in code.

**2.7 Git and GitHub**

* In distributed/decentralized VCS, you get the whole repository as private with all the logs etc. as compared to a centralized VCS which only lets you access the project.
* git add moves the changed files from the working directory to the staging area.
* git commit moves the files from the staging area to the repository.
* git init to create a repository.
* .git folder contains all the configs, commits, history, etc.
* git status to get the status of the branch.
* git remote -v to check if a remote repository has been configured.
* git remote add origin {link to repository} to configure the repository.
* If you have not added any new files and only modified the existing ones, then you can just use git commit -am which tells bash to first stage the changes and then commit. You don’t have to use git add .
* git branch to know which branch you are currently on.
* git checkout -b branch\_name for creating a branch. Remove -b to switch branches.
* git diff branch\_name for getting the differences between branches.
* git push -u origin branch\_name. -u indicates upstream.
* Conflicts occur when the same file is changed both in the master branch and the feature branch.
* git log --merge: produce the list of commits that are causing the conflict.
* git diff: Identify the differences between the states repositories or files.
* git checkout: Used to undo the changes made to the file, or for changing branches.
* git reset --mixed: Used to undo changes to the working directory and staging area.
* git merge --abort: Helps in exiting the merge process and returning back to the state before the merging began.
* git reset: Used at the time of merge conflict to reset the conflicted files to their original state.
* git branch -d branch\_name for deleting branch.
* You can stash your changes when you’re switching branches. Stash means that we’re not committing but putting aside the changes somewhere so that we can pick them up later.
* Closes #2 to reference issue in your pull request. When the pull request is merged, the issue will be closed.