Lec 1.1 - Intro

Concepts

To compile and run a JS program, enter ‘node programName.js’ into the terminal after entering ‘1531 setup’

- Node stands for NodeJS which is a program that compiles and runs JS

Strings in JS are a lot more flexible

- We can define strings using const or let sName = ‘anyString’;
- sName += ‘ abcd’; will append ‘ abcd’ to sName
- Strings don’t need to be null terminated ‘\0’

Collections are data structures

- Sequential collections → arrays
    - Values are referenced by their integer index (key) which represents their location in an order
    - We also use sequential arrays for c-style linked list type shit
    - Items in arrays do not have to be the same data type

Objects are mutable associative structs that may consist of many different types

- Used when we want a collection of items identified by string description not index

Code

console.log(…);

- Equivalent to printf → prints whatever … is in brackets, but does not require flags like %d %c
- Prints a newline after …
- console.log(string1, string2) will print string1 and string2 → console.log(`string1 is ${string1} and string 2 is ${string2}`); will also work

function fName(arg1, arg2, arg3, arg4)

- Defines a function called fName for you to write → doesn't need the return type or argument type to be defined

const vName = …;

- Declares an unchangeable variable called vName as …
- … can be an int, double, char or string
- Setting a variable to = undefined or = null means it is nothing
- JS handles data types for us so we don't need to define them

let vName = …;

- Defines a changeable variable called vName with value …
- Same as const except we can change the value of vName after

const sentence = `This is ${name} and I am ${age} years old`;

- Declares a string called sentence, where name and age are the names of other strings, and printing will print sentence with the actual strings of name and age
- E.g. if name = 'Felix' and age = '19', console.log(sentence) will print 'This is Felix and I am 19 years old'
- NOTE: in the line of code declaring sentence, those are backticks ` (left of 1 key), not apostrophes '

if/else if statements are the same, as are while/for loops

const array = [a, b, c, d];

- Declares a sequential collection called array with array[0] = a, array[1] = b and so on
- console.log(`${array}) prints all contents of array separated by commas
- console.log(`${array[0]}`) prints a, the 0th Item in array
- array.push(e); will add e to the end of array

for (const i in aName) { do sth }

- Does sth for all indices i in aName

```
const oName = {
        field1: ‘string’ ,
        field2: 989,
        field3: 1.1,
}
```

- Declares an object called oName with three fields; a string, an int and a double
- oName.field1 will access field1
- oName.field4 = ‘a’ will add a subvariable called field4 with value ‘a’
- console.log(oName) will print all 3 fields like so:
  { field1: ‘string’, field2: 989, field3: 1.1, field4: ‘a’}
- We can declare empty objects by using = {}; then filling it out later
- for (const key in oName) will iterate through all subvariables in oName, storing the subvariable name (e.g. field1) in key
- if (‘subvariable’ in oName) { do sth } → will do sth if ‘subvariable is the name of a subvariable in oName
- Object.keys(oName), Object.entries(oName) and Object.values(oName) returns an array of just the keys, just the entries, and just the values respectively
  - Entries are key-value pairs → will be printed as [[‘field1’, ‘string’], [‘field2’, 989], etc.]

Lec 1.2 - Git

Concepts

Git allows group work on code

- Tracks versions/changes to the code over time
- Multiple people can program at the same time
- All users have a full copy and backup of the work

Git has a tree-like structure

- Git is a collection of commits
- Each commit has one parent (it knows its parent), and can have multiple children called branches
- A branch is a pointer to a particular commit, and you 'merge' two separate branches to bring them together onto the same commit
- The 'master branch' is a pointer to (usually) the latest commit in the master copy of the repo
- A new non-master branch is a pointer to (usually) the latest commit in its copy of the repo → that latest commit knows its parent, and its parent knows its parent, giving us the whole 'branch' of commits that stems off from the master branch
- Merging just kind of makes the master branch point to the same commit as the branch it merges with???

Git Commands - Solo

git clone [url]

- Creates a local copy of the repo at the url and tethers the copy to the git cloud drive
- Repo → short for repository → sorta like a remote directory on gitlab
- Git url usually starts with git@...
- Need to clone every time you are working on a different computer

git status

- Shows you any new files added or any changes to your copy of the repo that aren't being 'tracked' by gitlab (they haven't been synced to the global repo)
- Also shows you any 'uncommitted' files it has recently started tracking

- Also shows you the number of commits you have done since the last push

git add fileName

- Adds fileName for gitlab to ‘track’ locally in the repo?
- You can add multiple files at a time → git add file1 file2

git commit -m “message”

- git commit is just a small save of your current working repo → a local save is git add + git commit → a local save is git add + git commit + git push
- After making changes, you should do local saves while working on code, and global saves when you are done for the time being
- -m means message → you can upload a message/note with your save, which will be shown alongside the changes you made with each commit on gitlab
- Adding -a before -m means it adds and commits all uncommitted files → local save everything in one command → BUT easy to make mistakes → you should just push and commit a file whenever you finish changing one file to see if the changes work

git push

- Synches your current committed copy of the repo to the gitlab cloud drive

git diff

- Shows any changes made to the repo, including small changes to files in the repo, since the last commit/push

git log

- Shows a history of all commits you have made as well as the messages (if any)

git pull

- Downloads the latest version of the repo on gitlab → syncs your local repo with the cloud repo

Git Commands – Team Usage

git checkout -b newBranch

- Creates a separate thread of working (branch) unlinked from the master/main branch
- Kind of like creating a copy of the master branch for you to work on yourself and then merge back to the master branch when you're done
- Commits on newBranch are local to newBranch → they won't show up on the master branch → no problem of colliding commits with teammates
- Moves you into newBranch
- -b signifies you're making a new branch called newBranch
- When working in a team, it is often good to name it Name/plans_for_changes → e.g. Felix/watch_fish
- On gitlab, you can see pushed commits in all branches

git checkout branchName

- Changes/switches from current branch to branchName

git push origin branchName

- Pushes an entire branch

git merge branchName

- Merges committed and pushed changes to the repo in branchName into the branch you are currently in
- When working in a team, merge to master branch using the gitlab UI (website), not the command → send a merge request → anyone else on the team can approve → you can click merge once they have and it merges your branch with the master branch
- If you attempt to merge conflicting branches (e.g. two non-master branches change one particular file), the first branch merged will be fine, but the second merge will be blocked until 'merge conflicts' are resolved

Resolving branch conflicts

- The author of the conflicting branch will have to
  1. Pull the newest repo in master branch

2. Run ‘git merge master’ in your branch → this will fail and tell you which file(s) the merge conflict is, as well as show, in your branch copy of the file, what the conflict is
    - Your changes are below ‘<<<< HEAD’ and above ‘=======’, the master version that is conflicting is below ‘=======’ and above ‘>>>> main’
3. Sort that shit out with the rest of your group, especially the author of the merged branch that caused your branch to conflict
4. If the conflicting branch is agreed on as the correct one, remove the changes from step 2, commit the change again, push the branch, a teammate needs to resolve conflict and approve on gitlab, you click merge

Lec 2.1.1 – Package Management

Concepts

Node Package Manager (NPM) is a tool that manages dependencies/modules/libraries for Javascript projects

- Automatically installed with NodeJS
- Can download external libraries from thenpmjs website
- package.json is where meta data about our project (including dependencies installed) is stored
- package-lock.json is where we store versioning information about dependencies to ensure everyone has the correct versions
- node_modules is where dependencies are installed locally → node_modules/pName is the directory where the implementation of functions provided by pName are stored
- ALWAYS commit changes to package.json and package-lock.json to git, but NEVER commit changes to node_modules to git → node_modules is very big

package.json

- We can add scripts in package.json
    - E.g. "test": "./node_modules/.bin/jest" would mean everytime you enter "test" in the terminal, it runs "./node_modules/.bin/jest"

Command

npm init

- Initialises npm for the current directory you are in
- Just hit enter for everything unless you want to change the name or know what you are doing
- Will rarely have to do this in 1531

npm install pName

- Adds pName to package.json and package-lock.json then installs all dependencies in them in your current repo/directory
- If no pName is input, npm just does the second part; reads package.json and package-lock.json files to install the appropriate dependencies
- Npm will also audit the packages and report the number of security vulnerabilities found

Lec 2.1.2 – Multi-File & Importing

Concepts

JS has a lot of built-in libraries

- They do not need to be installed
- In C, stdio.h is a built-in library

Code

import localName from 'lName'

- Imports a built-in library called lName, which will be called localName in the file
- Equivalent to the older code: var fs = require('fs')

localName.func()

- Calls a function called func that is in the imported library lName, parameters are passed into ()

export default fName

- Exports a single function fName
- Allows a single function fName to be exported, and thereby imported by other files
- Can also put 'export default' before function in the function declaration:

```
export default fName(parameters) {
        return null;
}
```

- If we want to export multiple things, we remove default and just list the functions we want to export in {}:

```
export {
        f1Name,
        f2Name,
}
```

- Think of exported functions like those declared in the .h file, and any non-exported functions that the exported ones depend on as helper functions in the .c implementation file import fName from ‘filePath’

- Imports an exported function fName in filePath → needs the path to the file fName is in → can use ./ or ../
    - If filePath has exported using ‘export default’, then we can name fName anything we want
- If we want to import multiple functions from filePath, we list them in {}:

  Import { f1Name, f2Name } from ‘filePath’
- If the file you are importing to already has a function called fName, there will be an error
- If f1 depends on f2, but you only want to use f1, you do not need to import f2

Lec 2.2 – Dynamic Verification

Concepts

We need to fkn write tests to know our shit works and that its safe and secure

- Software becomes unsafe when its design or implementation allow for unexpected or unintended behaviours
- Safety → protection from accidental misuse
- Security → protection from deliberate misuse

Javascript is a memory safe language

- We can't access uninitialized or unallocated memory

Static verification

- Checking your program when compiling → dcc saying shit broken or pointing out errors was static verification

Dynamic verification

- Checking your program works during execution
- White box testing → can see interface and implementation when testing
- Black box testing → can only see interface when testing
    - Abstracts details to focus on higher level understandings →
- Two categories of testing
    - Testing in the small
        - Unit tests → the testing of individual software components → usually just testing particular functions in isolation
    - Testing in the large → test larger parts together
        - Module tests → test specific modules
        - Integration tests → tests the integration of modules
        - System tests → tests the entire system
- We need to test for a lot of things
    - Invalid input
    - Unexpected data
    - Broken I/O
    - Undefined behaviour

Jest is a testing framework for JS that is installed from the NPM library

- Install by entering 'npm install --save-dev jest' on terminal in the repo you want to download on → this --save-dev option tells npm that only developers need this dependency → will be placed in a devDependencies section in packages.json, so non-developers will not download it

Good typical process:

1. Write function stub with comments about what the function does, its inputs and its outputs
2. Write tests
3. Add, commit, merge
4. Implement the function
5. Test
6. Add, commit, merge when satisfied

Code

```
describe('description', () => {

        test('test category description', () => {

                const variable = Actual tests (e.g. function call);

                expect(variable).toEqual('expected output');

                const variable = More actual tests (e.g. function call);

                expect(variable).toEqual('expected output');


        });

        test('some other category', () => {

                expect(functionCall(a, b)).toEqual('expected output');

                expect(functionCall(c, d)).toEqual('expected output');

        });

});
```

- Typical code to test stuff
- Describe just lets you put some words to help you understand your tests when you write them and when you run them
- You can have as many describes as you want, and they are nested → cannot put tests within tests, however

expect(rName).toSth(‘expected output’)

- expect function compares the result stored in rName (or a direct function call in place of rName) to something
- The .toSth is an option
    - .toEqual(‘expected output’) → checks if rName equals ‘expected output’
    - .toBeLessThan(number | bigint) → checks if rName is less than number
    - .toContain(item) → checks if rName contains item
    - .arrayContaining(A) → checks if an array rName contains the array or item A
- Just look at the jest library to see what options and functions there are

./node_modules/.bin/jest testName.js

- Runs the tests written in testName.js
- If no testName is given, it runs all tests
- Will tell you how many tests passed/failed, and also shows all failed tests with their output and expected output

Lec 3.1.1 – Data Interchange

Concepts

Standard interfaces are necessary for data to be transferred and stored between different applications

- Data interchange formats are extremely simple markup languages that are collections of arrays, objects, numbers, strings, Booleans, and they enable this standard interface
- Instead of writing all permutations of bindings to convert from one code language to another, we write a binding from a language to a standard interface, and all languages that have a binding to the interface can access data from another language

JavaScript Object Notation (JSON) is a format made up of braces for objects, square brackets for arrays, numbers, and non-numeric items wrapped in "quote" marks

- JSON.stringify(DATA_STRUCTURE) will return a stringified copy of data structure → converts it to JSON format where object keys and shit are wrapped in "quote" marks

YAML (YAML Ain't Markup Language) is another modern interchange format

- A dash – means it is an item in a list/array, indentation matters (how nested the item is in arrays/lists), no curly braces or square brackets

eXtensible Markup Language (XML) is another older interchange format

- Very annoying to read, won't use in this course at all, just know it exists

Code

Lec 3.1.2 – Continuous Integration

Concepts

Continuous integration is the practice of automating the integration of code changes from multiple contributors into a single software project

- That is, it helps make merges into master more frequent and stable
- Involves a series of operations that are executed on any commit pushed to the repo
- Gitlab handles continuous integration with a .gitlab-ci.yml file within the root of your git repo, which contains a set of instructions called a 'pipeline'
    - It will have a list of the stages, then instructions sets where each line in the script is a shell (terminal?) command

When a commit is pushed, all of the code int hat commit is taken by another computer (runner) and has the .gitlab-ci.yml instructions run on it

- Basically, runners are computers that solely run pipelines to check shit
- Gitlab does not have built-in runners, but course sets them up for us

Code

```
cache:

    paths:

        -   node_modules


stages:

    - stage1

    - stage2


Instructions1:

    stage: stage1

    script:

        -   echo 'hi'
```

```
instructions2:
    stage: stage2
    script:
        - npm run test
```

Lec 3.2.1 – Type Script

Concepts

We can’t expect everyone to always use a function correctly

- We must handle invalid/unexpected input
- Logic errors are errors that do not cause a crash → they are an error in your logic → we need assert-like checks to force the program to crash so that we know there is an error → an even better way is to use type safety

Type safety prevents mismatches between the actual and expected type of variables, constants and functions

- C is type safe → types must be explicitly declared and the compiler checks that they are correct
- Typescript is a language built on top of JS → it checks types in your TypeScript program and outputs JS that is then runnable with node
    - Allows you to specify types in JS → if the specified types don’t match up, will cause an error like in C
    - Typescript files are .ts

npm install –save-dev typescript ts-node → package for TypeScript

- TS can sometimes infer types → e.g. if a and b are only used for multiplication, it infers that a and b are probably numbers
- In tsconfig.json there is a section called “compilerOptions” that lets you toggle what the compiler counts as an error
    - noImplicitAny → important → all variables must have their type specified → will be toggled on for tsc but not ts-node in 1531
    - noEmit → if toggled off, tsc will create a .js file that is the .ts file with all TS stuff stripped away
- node_modules/.bin/ts-node myProgram.ts
    - Interprets TS into JS and runs it
    - Takes a bit longer to run
- node_modules/.bin/tsc myProgram.ts →
    - tsc = TypeScript Checker/Compiler → this command checks that types match without running myProgram
    - Will compile your TypeScript and output a JavScript file

- In package.json, we can put scripts so that we don't have to type out the whole node_modules/.bin/………. Shit everytime we want to run or check out code, we can just type npm run tsc or npm run ts-node myProgram.ts
- We can also add tsc to our pipeline so that types get checked before being pushed

Code

if (vName instanceof 'dataType') { do sth }

- Does sth if the variable vName stores data of type dataType
- E.g. num instanceof 'number' will return true
- E.g. num instanceof 'string' will return false

console.error('string')

- Prints 'string' to stderr

TYPESCRIPT

let vName: datatype = val

- Declares a variable called vName that is of type datatype and has value val
- Basic/Primitive dataTypes are number, string (chars are 1 symbol strings), Boolean
- Complex data types include arrays and objects
- any is a special dataType → shit can be anything → don't use permanently
- We can also do things like vName: boolean | number if vName can be a Boolean or a number
- If you do const vName = 18, TS can infer that it is a number

let arr: Array<dataType> = []

- Can also do let arr: dataType[] = [], IF dataType is a primitive data type or a typedefed one → an array of objects needs the object to be typedefed
- Creates an empty array of dataType called arr
- TS usually can't infer anything with arrays

function fName(var1: number, var2: string): boolean {

- Function called fName that takes in a number and a string and returns a boolean

- Don’t always need to specify return types
- If we had var2?: string, it means var2 can be a string or undefined → users do not have to input a string, if no string is input var2 is defaulted to undefined → need to handle if it is undefined → note, undefined will hold false for logic

type tName = dataType

- We can give an alias ‘tName’ to dataType → same shit as typedef in C
- dataType can be a hybrid (e.g. string | number),
- We can also typedef tName to string literals → type tName = ‘a’ | ‘b’ | ‘c’ → gives us more ways of defining options than true or false

```
type Object = {
        var1?: number;
        var2?: string;
        var3: boolean;
}
```

- Defines an object data type called Object which contains a number or undefined, a string or undefined, and a Boolean
- NOTE: we put ; not , after every line
- Subvariables that can be undefined do not have to set upon creation of Object
- const vName: Object = { … } declares a variable of type Object

Lec 3.2.2 – Linting

Concepts

Linting refers to software that statically analyses your code and automatically makes adjustments to fix style

- Linters also have some basic semantic checking for possible bugs
- They don't help name variables

eslint is a popular linting tool for JS → npm install –save-dev eslint

- Can detect errors, warn of potential errors and check against good style conventions (can configure how strict the style is)
- Can also automatically fix issues with your style
- Style rules are written in a .eslintrc.json (or .js) file → this is provided to us in 1531
- ./node_modules/.bin/eslint myProgram.js
    - Checks myProgram.js for any problems, reporting any that do come up
- ./node_modules/.bin/eslint --fix myProgram.js
    - Eslint checks and fixes your myProgram.js
- /* eslint-disable */ → tells eslint to ignore everything
- // eslint-disable-next-line → tells eslint to ignore any errors on the next line
- Can also add eslint shit to pipeline for style checks

Code

Lec 4.1.1 – Advanced Functions

Concepts

First class functions are functions that can be treated as a variables

- A first class function can be returned or input into another function, or can be assigned to a variable
- We can also pass functions as arguments → like function pointers in C

Anonymous functions are first class functions we don't name because we only ever really use it as a helper in one other function

- If a function takes in a number num and a function func as its arguments, we can directly write func in the function call if func is short → if func is long, best to keep separate
- The alternate arrow => function format is nice for writing anonymous functions since it is very short for one line functions
- Keep shit readable tho

Higher order functions are functions that return a function

- They create and return another function
- We can use these to create functions that have almost identical code without repeating that near-identical code every time

Code

const fName = (arguments) => {

- Alternative function prototype → function fName(arguments) {
- If fName's body is one line, we can just put that next to the arrow =>
    - e.g. const sum = (a: number, b: number) => a + b;

function doSth (var1: any, func: fType) {

- Function prototype that takes in a var1 of any data type, and a function of fType
    - fType should be a typedefed function → e.g.

    - This allows whatever function is input as func to be called within doSth → can avoid code/logic repetition

type fType = (arguments with types specified) => return type

- All functions that take in the same type arguments and return the same types are functions of type fType
- E.g. type Math = (num: number) => number;
    - Functions of fType Math take in a number and return a number

function xSum(x: number, a: number, b: number, (a: number, b: number) => a + b)

- xSum calculates the sum of a and b, multiplied by x
- (a: number, b: number) => a + b, is an anonymous function

arrayEx.map(fName)

- Returns a copy of arrayEx where each element has been modified by fName
- E.g. if arrayEx was an array of numbers, and fName is a function that doubles numbers, the returned array would be arrayEx with all elements doubled
- fName can be an anonymous function → shit can get even more condensed and simpler in terms of style

arrayEx.filter(fName)

- Returns a copy of arrayEx with only the elements that are 'true' according to fName
- fName should return a Boolean
- e.g. if arrayEx was an array of numbers, and fName is a function that returns true if a number is even, the returned array would be arrayEx with only the even elements

arrayEx.reduce(fName, num)

- fName takes two arguments → the previous result and the next element → starts with num and the first element, then does the result of that with the next element and so on
- Given arrayEx, it returns a single output summarising it
- E.g. if arrayEx was an array of numbers, and fName is a function that sums two numbers, arrayEx.reduce(fName, 0) would effectively give the sum of the array

Lec 4.1.2 – HTTP Servers

Concepts

Definitions

- Network: a group of interconnected computers that can communicate
- Internet: a global infrastructure for networking computers around the entire world together
- World Wide Web: a system of documents and resources linked together, accessible via URLs
- Web falls under internet, internet falls under network

Network Protocols

- Communication over networks must have a certain structure so that everyone can understand → we call these structures network protocols
- Different protocols are used for different types of communication
- HTTP is the primary protocol used to access URLs in your web browser
- The 'Express Server' library on npm allows us to run our own HTTP server with NodeJS

Application Programming Interface (API) refers to an interface exposed by a particular piece of software

- Interface for a blackbox → lets you use things without knowing how they work
- A RESTful API is an API that uses HTTP requests to GET, PUT, POST and DELETE data → 4 CRUD operations that describe the 'nature' of different API requests
  - GET → Read data
  - PUT → Update data
  - POST → Create data
  - DELETE → Delete data
- Different CRUD opertions have different approaches for input → output is the same
  - GET or DELETE → uses req.query to get input
  - PUT or POST → uses req.body to get input
  - Output should always be packaged up as JSON → JSONified string n shit
    - res.json JSONifies shit for us
- REMEMBER: the CRUD operations are from the perspective of the user??

GET or DELETE operations

- Data is passed in as key-value pairs in the URL → localhost:3000/path?fish=shark&num=1
- ? indicates query → key-value pairs get converted to an object stored in req.query → req.query.fish === shark
- & is used to list data
- Data is always a string, so need to use parseInt or whatever
    - May sometimes require 'as string' → parseInt(req.query.num as string) === 1
- The passed in data will be stored in/deleted from req.body

PUT or POST operations

- Data is stored in req.body as an object → if we have {"x":2,"y":1} in req.body, we can access that using req.body.x and req.body.y

For this course, all web servers run on HTTP using express() will be on localhost:

- localhost:3000 → web server at port 3000 → type into a web thing on vlab or whatever

We can talk to a web server as a client by using a web browser, API client or npm library

- Web browser → using the vlab web browser, just type localhost:3000 and any paths or data you want to input and shit
- Installing API client Advanced Rest Client → google "ARC client googel chrome addon" → install the addon and open it → follow demo in lecs to configure
- npm package sync-request allows us to send RESTful API requests using code
    - npm install sync-request
    - import request from 'sync-request';
    - This shit allows us to test our code → the others only let us debug

Testing we use jest with sync-request

- We should write functions in the test file to avoid having annoying repetitive code regarding requests and JSONifying shit and getting the body
- 1:05:50 of 22T3 lec 4.2

Code

const app = express();

const port = 3000;

- express() creates and returns an instance of a server to app
- port = 3000 sets app's port to 3000 → can choose any number greater than 1023, less than 65000 → greater than 3000 is a good choice
- App would run at url localhost:3000

app.use(express.text());

- Some shit that's needed for the data of many requests to be interpreted when using expresss
- If we change text() to json()
    - We can use shit like res.json to send JSONified objects
    - We do not need to JSON.parse things in req to convert them to JS/TS objects

app.CRUD(path, (req, res) => {

do sth

});

- Uses the specified CRUD (get, put, post, delete) operation on app at the given path (string) → if path was '/apple' and port was 3000, it would do sth at localhost:3000/apple
- Path must be a string starting with '/'
- req = request → any data passed is stored in req as an object??
- res = respond → in response to someone accessing this url, do some shit

app.post(path, (req, res) => {

do sth

});

- Uses POST operation
    - Data is stored in req.body →

app.listen(port, () => {

Code and shit;

});

- Starts running the server for app at port port, doing whatever → for 1531 we work on our own devices, so its always localhost:port

JSON.stringify({ msg: str });

- Turns the object into a string → e.g. if it was { msg: 'fuck' }, JSON.stringify would return the string '{"msg":"fuck"}'

res.json({object})

- Sends shit in JSON

USING sync-request LIBRARY

const res = request('CRUD', 'url', { object })

- Does the specified CRUD operation at the input url → both CRUD and URL are input as strings → CRUD operation string is 'GET', 'PUT', 'POST', 'DELETE'
- Stores the result of that CRUD operation in res
- For POST, the object is usually of the form

```
        {
                json: {
                        data1: 'fucking',
                        data2: 'lost'
                },
        }
```

  and is whatever data we want to create/send to the server → note that labelling the object json just saves us from having to JSONify shit and use the header and body labels

- For GET, we can input data through the URL using ?fish=shark&num=1, or we can also pass in an object like we do for POST → we label the object qs (query string), not json tho
- In ADDITION to json/qs objects, we can also pass in a headers object → this input will be stored in res.headers → can pass headers for all CRUD ops → headers are encrypted so we send important shit thru there → e.g. tokens

- REMEMBER: everything output from web stuff is a JSON package → need to JSON.parse and shit

res.field

- Returns the field field in res → res.body, res.header

Lec 5.1.1 – Persistence

Concepts

Data is shit like stats n input, and information is the insights we can gain from those stats n shit

- In software, 'data' usually refers to the data layer, which is the part of software focused on storing data and maintaining a state for a longer term

In 1531, persistence refers to storing data to disk

- Officially, it is when the program state outlives the process that created it → storing the state as data in computer data storage → after the process ends, the program storing the data still runs
- We can do this by loading and saving on/to a file
  - Load: on server start, open a JSON file and set the global variable to be the parsed JSON object
  - Save: based on a timer or at the end of every server request, JSON stringify your global variable and store it in a file

Persistence for 1531 involves writing and reading our global variables (dataStore) to/from a file

- import fs from 'fs' → 'fs' library does not need npm
- To save data to a file, write JSON.stringify(data) to a file
- Then to get that data when you next start your server, just read from the file
  - Since we wrote to it as a JSON string, we will need to read the data as a string, then use JSON.parse
- NEED to make sure the file you are reading from/writing to exists

Code

fs.writeFileSync(fileName, data)

- Writes data to fileName
- Data needs to be a string?

fs.readFileSync(fileName)

- Will return the contents of the file fileName

fs.existsSync(fileName)

- Returns true if fileName exists, and false if it doesn't

LECTURE CODE:

```
let data = {
        Whatever data structure shit you want
}
```

- Let, because we want to be able to reassign it in the next step if we have something saved in fileName
- If fileName does not exist, the above defined data does not change → it continues from its initial state

```
if (fs.existsSync(fileName)) {
        const dbstr = fs.readFileSync(fileName);
        data = JSON.parse(String(dbstr));
}
```

- Load function → not actuall a function, should just be at top of whatever so that data is retrieved from fileName whenever server starts up, IF fileName exists
- If fileName does not exist, data is just in its initial state

```
const save = () => {
        const jsonstr = JSON.stringify(data);
        fs.writeFileSync(fileName, jsonstr);
}
```

- Save function → saves the global variable data to fileName

Lec 5.1.2 – Exceptions

Concepts

We want to be able to throw and catch exceptions so that we can handle error cases better (more elegantly) at runtime

- Exceptions are actions that disrupt the normal flow of programs → often represents an error being thrown
-

Code

throw new Error(errorString)

- Will throw an error to the top → will crash the program immediately and print errorString to stderr
- Will look for something to catch the thrown error → if not in the current function, will kill the current function then look for a catch in the parent function, and so on until it either finds a catch or terminates the whole program

```
Try {
        Do something
} catch (err) {
        console.err(`errorString ${err}`);
}
```

- Runs the code (Do something) in the try block, and if an exception is thrown when trying to run it, the catch(err) will immediately catch the thrown the exception and the program switches to the catch block → prints errorString alongside the caught error and does whatever else is in that block
- err is the entire bunch of lines that shows if the thrown exception just causes the program to crash (no try-catch block) → err.message is the errorString thrown
- If the code does not return or exit in the catch(err) block, it just continues like it would for an if else statement
- We can put a try-catch block of code inside a while loop with a conditional variable → set conditional variable to false, then while false run the try-catch block, if the try is successful, set the conditional variable to true, this will break the loop

```
expect(() => funcCall()).toThrow(errorString)
```

- Jest matcher to handle thrown exceptions
- Needs a function in the expect() bracket, NOT a result

Lec 5.2 – Desgining for Maintainability

Concepts

We want software to be maintainable → resistant to breaking when things change over time

- Testing, system design and code design help improve software maintainability
- Testing → Having a way to verify your code's correctness makes it easier to change it without worrying that something breaks
- System design → Planning systems to make sense at a very high conceptual level
- Code design → Thinking about code at a high and low level of detail in terms of what makes things resilient will make your code more resilient and adaptable to changes in the future

Code design → planning out how code should be written, and written elegantly

1. Is there one source of truth for this?
   - Basically, is your shit repetitive and hard to change
   - E.g. using constants so that variables do not need to be changed everywhere
   - E.g. using helper functions to avoid repeating code everywhere and changing all instances of that code
2. Is this as simple as possible?
   - Keep shit short and easy to read/understand
   - Take advantage of libraries and JS functions
3. Is this over or under-designed?
   - Over-designed has very complex abstractions to maintain for trivial changes
   - Under-designed
4. Are related modules kept close together and unrelated modules kept far apart?
   - Coupling is the degree of interdependence between software components → want related components to be tightly coupled, and unrelated components to be loosely coupled
   - The more software components are connected, the more changes to one component may break another
5. Am I speculating about how necessary this is?
   - Sometimes best not to add a feature/functionality until you actually know you need it
   - Avoid writing low level utility functions that are rarely or never used
6. Does this follow standard conventions?
   - Standard convention allows people to easily understand each others code

Lec 7.1.1 – Code Coverage

Concepts

Test coverage refers to how much of the feature set is covered with tests

Code coverage is a measure of how much code is executed during testing

- Often a percentage of statements (lines) executed during testing → gives us an idea of how much of our code is executed by the tests, and highlights what hasn't been executed
- Add "test": "jest –coverage" to the scripts in package.json → will show you the code coverage of a test for each file tested and any untested lines → you want 100% coverage
  - Creates a directory called coverage that contains a directory called lcov-report, which contains a file index.html → open this with firefox (tigerVNC, dk how to do on vscode) to see a more detailed coverage report, and a copy of your file with the unused lines highlighted
  - Will also show coverage for any dependent files → e.g. our own files we imported functions from

Branch/Edge Coverage is a measure of how many branches (routes in code from while, if else statements etc.) are covered by your code

- Done automatically with jest? → not really a big thing for 1531 bc it doesn't work that well without a lot of other shit
-

Code

Lec 7.2.1 – Conceptual Modelling

Concepts

Models are simplified representations that assist in understanding something more complex

- Structural conceptual models emphasise the static structure of the system
    - ER Diagrams → database design → drawing what a datastore should contain
- Behavioural conceptual models emphasise the dynamic behaviour
    - Case diagrams → flow chart shit

State diagrams are a diagrammatic representation of the state of something

- Circles (or any shape) represent different states, labelled arrows represent the transitions between each state
- Used to describe state machines that are made up of a finite number of states, and the machine can be transitioned from one state to another through an interaction
    - Software is a state machine

Code

Lec 7.2.2 – Deployment

Concepts

Deployment refers to the making of software accessible to people

- Continuous Delivery → periodic deployment or "ship", and if the program passes the pipeline, someone signs off on the ship and deploys the program
- Release methods for deployed code usually has 3 core tiers:
  - Dev: released often, available to developers to see their changes in the deployment
  - Test/staging: Released close to release???
  - Prod: Released to customers, ideally as quickly as possible

DevOps

- Category of roles focusing on skills that overlap between development, QA and operations → merging of roles and multi-skilling

Maintenance: After deployment, analytics and monitoring tools must be used to ensure that the platform is used and remains in a healthy state

- Monitoring preserves user experience (monitors errors, warnings and shit that affects performance or uptime), and enhances user experience (analyses users and their interactions)
-

Code

Lec 8.1.1 – Requirements

Concepts

Before we code anything we need to know how to test and design it, and to do that we need to know what the requirements of the system we're building are

- Requirement: a condition or capability needed by a user to solve a problem or achieve an objective
- Functional Requirements specify a specific capability/service that the system should provide → what the system should do
- Non-functional Requirements place a constraint on how the system can achieve that → performance related → e.g. time complexity requirements

We derive the requirements by following 4 steps:

1. Elicitation (extraction) of raw requirements from stakeholders
   - Market research and focus groups and shit
2. Analysis of requirements
   - Identify dependencies, conflicts, risks, and establish relative priorities
   - Usually done through 'User stories' and 'Use cases'
3. Formal specification of requirements
   - Break it up into functional and non-functional requirements, and make it formal and clear cut
   - Make sure requirements aren't over or under detailed/specific
4. Validation of requirements
   - Go back to step 1, and check that the requirements are correct

Challenges to requirements engineering

- Stakeholders not knowing or changing their mind about what they want
- Developers not understanding the subject domain
- Going into the details and solutions too early

User stories are a method of requirements engineering used to inform the development process and what features to build with the user at the centre

- User-Centred development type shit
- "As a <type of user>, I want <some goal> so that <some reason>"

- Features: non-technical language, describes problems not potential solutions, independent of other user stories, negotiable, small, valuable, testable

User Acceptance Criteria is a method by which we can 'test' whether a user story has been satisfied

- Break down the user story into a criteria (in plain English) that must be met for the user to accept → can be refined before using
- Acceptance criteria should be moderately specific, not too technical, written before design and testing
- Acceptance tests are blackbox tests that are performed to ensure the acceptance criteria is met

User Cases represent dialogue between the user and the system, with the aim of helping the user achieve a specified goal

- User initiates actions and the system responds with reactions
- They consider systems as black box and are only focused on high level understanding of flow → if user does this, system should do that, disregard the how (implementation)

Validation is a set of activities ensuring and gaining confidence that a system can accomplish its intended goals and objectives

- Verification checks that the system has been built right, whilst validation checks that the right system has been built
- Usually validate software by letting real customers use it or QA employees test it and go through a checklist to make sure the software is behaving as intended and solves the original problem

Code

Lec 9.1.1 – Auth

Concepts

Authentication vs Authorisation

- Authentication is the process of verifying the identity of a user
- Authorisation is the process of determining an authenticated user's access privileges

Authentication

- Simple way is to assign an ID token to them on registration, and set their username and pwd → in theory only they know their username and pwd
- Encryption and hashing are two processes by which plaintext information is transformed into seemingly random strings of characters → more secure way of storing/hiding info
  - Encryption is reversible → less secure
  - Hashing is irreversible → more secure
- We can store hashed pwds in our server to make it more secure → when they registr, hash their pwd and store that instead, then when they login, hash the input pwd and check that it matches → if the pwds are the same, their hash is the same
- A rainbow table is a table that charts a bunch of permutations of hashes to find the hash of a bunch of pwds or whatever else is stored and hashed
- Taking it further, by adding some secret key/string to the pwd then hashing it together, we can prevent the creation of rainbow tables → unless mfs rly bothered n try all permutations of secret key/string as well
- FOR ITERATION 3 → If we have sessionIds, we should also store a sessionHash, which is the sessionId + SecretString hashed together → then we need sessionId and sessionHash to both match for a login to work

Code

crypto.createHash('sha256).update(plaintext).digest('hex')

- Requires import crypto from 'crypto'
- sha256 is a hash function → we can pass in other hash functions in the nodeJS stuff instead
- Hashes plaintext

Lec 9.2.1 – Software Complexity

Concepts

We need to be able to analyse how complex software is

- Accidental complexity → complexity that is not inherent to the problem → little overheads and shit? → e.g. parsing data in specific formats
    - Hard to fully remove → can reduce a lot by using libraries and shit
- Essential complexity → complexity inherent to the problem → the main shit → e.g. the cost of doing n specific tasks
    - Can only be managed with good software design
- Coupling is a measure of how closely connected different software components are
    - Loose = not very connected, tight = highly connected
        - E.g. backend should be loosely coupled from frontend
    - Loose coupling is good
- Cohesion is the degree to which elements of a module belong together
    - Elements belong together if they're somehow related → auth functions in auth file
    - High cohesion is good
- Cyclomatic complexity is a measure of the branching complexity of functions
    - Count the number of linearly-independent paths through a function
        1. Convert the function into a control flow graph
        2. Calculate the value of the formula $V(G) = e - n + 2$, where e is the number of edges and n is the number of nodes → e is the number of lines, n is the number of stages
    - The higher V(G) is, the worse the cyclomatic complexity is

Code

Lec 9.2.2 – Git Undo

Concepts

We can undo things or change history on git

Git reset

git commit amend

-

Code

git reset --hard HASH

- HASH is the commit hash → the string of random chars after "commit" when you run git log
- Will do a hard reset to that commit, ignoring everything you have done since then
- If you have the HASH of a commit more recent than the one you hard reset to, you can use this command to go back to that one
- Needs a forced push cause git likes new shit → git push --force → we can't actually do this bc we can't remove protections on our student repos

git reset --soft HASH

- Keeps all of your current code the same, but just changes what commit you're pointing to → Git thinks you are at HASH commit, when really you're at the newest one → nothing changes beside what commit Git thinks you're at

git commit --amend – m "commit msg"

- Allows us to update our previous commit name/msg
- Instead of adding another commit to the history, this will replace the most recent commit

Lec x.y

Concepts

Code

Lec x.y

Concepts

Code

Lec x.y

Concepts

Code

Lec x.y

Concepts

Code