Sr. Software Engineer (SSE)

ABSTRACT

This is one of the subject from my personal notes series named “Coding-With-Arqam” that I am developing from the start of my professional development career.

Subject

JavaScript + NodeJS

JavaScript

--> General Points:

-> Node.js is Asynchronous, single threaded.

-> JavaScript is always synchronous and single-threaded.

-> JavaScript is only asynchronous in the sense that it can make, for example, Ajax calls.

The Ajax call will stop executing and other code will be able to execute until the call returns (successfully or otherwise), at which point the callback will run synchronously.

-> Scripting Language meaning it is rendered when the code is loaded.

-> Old name: LiveScript

-> The programs in this language are called scripts.

-> Scripting language. Not a Compiled language like C#, Java, etc.

-> JavaScript can execute not only in the browser, but also on the server.

-> Programs = JavaScript engine.

-> The browser has an embedded engine sometimes called a "JavaScript virtual machine".

-> Different engines have different code names like V8 (in chrome and opera), spiderMonkey (Firefox).

-> JavaScript was initially created as a browser-only language, but is now used in many other environments as well.

-> Objects can store properties.

-> In JavaScript we can only inherit from a single object.

-> The JavaScript language was initially created for web browsers.

-> Since then it has evolved and become a language with many uses and platforms.

-> When the browser loads the page, it "reads" (another word: "parses") the HTML and generates DOM objects from it.

-> JS is not an object oriented. Classes in Js are just functions.

-> JS = frontend + backend language.

-> "isNaN" = is not a number.

-> JS is faster than ASP(Active Server Pages).

-> "negative infinity" = negetive number / 0

-> "this" keyword = refers to the object from where it was called.

-> Triple equal(===) = compares without any type conversion.

-> JS supports automatic type conversion.

-> Loops: For, While, do-while

-> Variable typing =

```
        i = 10;
        i = "string";
```

-> 3+2+"7" = 57

-> JavaScript only has a single type for numbers(floating numbers).   e.g: 98 === 98.0

-> JavaScript has no type for characters(only string type). e.g: 'abc', "abc"

-> let x; // declaring x (mutable)

*-> let y = 3 * 5; // declaring and assigning.*

-> const z = 8; // declaring z (immutable).

-> Ordinaryfunctiondeclarations: function add1(a, b) { return a + b; }

-> Arrowfunctionexpressions: const add3 = (a, b) => a + b;

-> Arrays are also objects.

-> The grammatical category of the variable and property name is called the identitfier.

-> Casing Styles: camel case, underscore case, dash case.

-> Capitalization of names:

-> Lowecase/camel: functions, variables, methods, Constants

-> Uppercase/Title: Classes, Constants

->If the name of a property of an object starts with an underscore then that property is considered private.

-> Where to put semicolons?

-> at the end of the statements but not with the statement that ends with curly braces,

-> Reserved Words:

-> await, break, case, catch, class, const, continue, debugger, default, delete, do, else, export, extends, finally, for, function, if, import, in, instanceof, let, new, return, static, super, switch, rhis, throw, try, typeOf, var, void, while, with, yield.

-> Automatic semicolon insertion(ASI)

-> let = mutable variable declaration. Reference can be changed.

-> const = immutable variable declaration, must initialize. Reference cannot be changed.

-> Same variables cannot be declared with in same scope called Shadowing variables

-> local variables (bound variables), global(free variables)

-> Floating point numbers can only be expressed in base 10.

*-> 2 ** 4 = 2^4 = 8*

-> const arr = [4]; arr[0]++; means -> arr, [5];

-> for (const n of numbers) { if (n < min) min = n; } // For-Of Loop

-> x === Infinity // assigning an infinity value

-> Tab: '\t'

-> const mySymbol = Symbol('mySymbol');

-> if (value) {}, if (Boolean(value) === true) {} -> both are samefeedback

-> continue onlyworksinside while, do-while, for, for-of, for-await-of, and for-in.

*-> eval('2 ** 4') -> 16 // given a string that will be evaluated.*

*-> import * as myMath from './lib/my-math.mjs';*

-> date.now() // returns the number of milliseconds elapsed since January 1, 1970, 00:00:00 UTC

-> new Date() // current date

-> pop() // removes the last element of an array.

-> shift() // removes the first element of an array.

-> unshift() // adds a new element to an array (at the beginning). "unshifts" was the older method.

-> splice(0, 1); // Removes the first element of fruits.

-> arr1.concat(arr2, arr3) // concates three arrays.

-> days started from 0 and that is sunday.

-> default ip for HTTPs (SSL) = 443 and for HTTP = 80

-> A tidy number is a number whose digits are in non-decreasing order.e.g: Input : 1234 Output : Yes Input : 1243 Output : No

-> JavaScript is object-oriented, but is not a class-based object-oriented language like Java, C++, C#, etc.

-> Thats why not implements OOP concepts. Classes in it are just like functions.

-> JavaScript, inheritance is supported by using prototype object.

-> Some people call JS inheritance as "Prototypal Inheriatance" and some people call it "Behaviour Delegation".

-> There is no private, static etc variables in JS like in C3, java etc.

-> Asynchronous processing started with 'callbacks', then came Promise and then async and await.

-> All of the I/O methods in the Node. js standard library provide asynchronous versions, which are non-blocking, and accept callback functions.

-> Unshift() and shift() work in exactly the same way as push() and pop(), respectively, except that they work on the beginning of the array, not the end.

-> Async function = returns the promise.

-> In Node Application, any async function accepts a callback as the last parameter and a callback function accepts an error as the first parameter.

-> ECMAScript 2015 Read More: JS ES6.

-> Declare all constants in a contant.js file.

-> Declare all errors in a error.js file.

-> Loosely-typed language.

-> No other platform comes close to Node.js — 3 Billion downloads

-> NPM aka Node Package Manager is the world's largest software registry, with over 3 billion downloads a week.

-> NodeJs is async and sync behaviour is acheived through async-await etc.

-> JavaScript is a case sensitive language.

-> Mocha was the real name of javascript.

-> "Framework" calls your "Code" and then your "Code" calls the "Library".

-> "Framework" contains the "Library".

-> iterables(arrays / objects / strings)

-> NaN !== NaN

-> err should be the first argument and cb should be the second.

-> Any code that you want to be accessible in another file can be a module.

-> Most programming languages that were not traditionally async today do have 3rd party libraries that implement ways to call asynchronous code.

-> PHP is a single threaded. can be onverted to multi-threaded using "pthread" library.

-> In Python, by default programs run as a single process with a single thread of execution; this uses just a single CPU (CPU core).

-> Status Code 304: A 304 Not Modified message is an HTTP response status code indicating that the requested resource has not been modified since

the previous transmission, so there is no need to retransmit the requested resource to the client.

-> Require statement are cached.

-> Reloading is making another http request to the webhost's server. It returns html for your browser to load onto the page.

-> Rerendering is the act of changing, adding, or removing existing html on the page that has already been served to the browser.

--> How do engine works:

-> The engine (embedded if it's a browser) reads ("parses") the script.

-> Then it converts ("compiles") the script to the machine language.

-> And then the machine code runs, pretty fast.

--> What makes javascript unique:

-> Full integration with HTML/CSS.

-> Simple things are done simply.

-> Support by all major browsers and enabled by default.

-> JavaScript is the only browser technology that combines above three things.

--> Code Editors:

-> There are two main types of code editors: IDEs and lightweight editors.

-> IDE Editors:

-> Integrated Development Environment.

-> It's not just an editor, but a full-scale "development environment.

-> Loads more data on start of the project.

-> E.g: Visual Studio Code (cross-platform, free), WebStorm (cross-platform, paid).

-> Read-Eval-Print-Loop

-> Lightweight Editors:

-> Not as powerful as IDEs, but they're fast, elegant and simple.

-> They are mainly used to open and edit a file instantly.

-> *E.g: Atom, Visual Studio Code, Sublime Text, Notepad++, Vim, Emacs*

--> Developer console:

-> Chrome Console:

-> Can access through "inspect element" or "F12".

--> Script tag:

-> JavaScript programs can be inserted into any part of an HTML document with the help of the <script> tag.

-> Internal Script:

-> E.g: <script> alert('Hello, world!'); </script>

-> External Script:

-> E.g: <script src="/path/to/script.js"> </script>

--> Code Structure:

-> Statements, Semicolons, Single line comments, Multi line comments

--> Strict Mode:

-> strict mode VS default mode.

-> ECMAScript version 5.

-> It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.

-> Modern mode.

-> With strict mode, you cannot do tasks like, cannot use undeclared variables.

-> When it is located at the top of a script, the whole script works the “modern” way.

-> They eliminates some JavaScript silent errors by changing them to throw errors.

-> Actually it is for strict coding standards.

-> Example:

-> If we declare a variable without datatype (like global variables in angular) i.e: temp = 'var', this will give error but this will not give error if we have not used strict mode.

--> Variables:

-> var (global scope, old)

-> let, const (funtional scope)

--> Data Types in node js:

-> number , boolean , string , and object

--> Null VS Undefined:

-> Null:

-> No data type.

-> Itself a data type.

-> In JavaScript, null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.

-> It’s just a special value which represents “nothing”, “empty” or “value unknown”.

-> Undefined:

-> No data type.

-> Itself a data type.

-> The meaning of undefined is “value is not assigned”.

-> If a variable is declared, but not assigned, then its value is undefined.

-> Normally, we use null to assign an “empty” or “unknown” value to a variable.

-> *Eg: if we use a var after decalring, error will come that var is not initialized. but if we put null as the value, we can use it now.*

--> Objects and Symbols:

-> The object type is special.

-> All other types are called "primitive" because their values can contain only a single thing.

-> Objects are used to store collections of data and more complex entities.

-> The symbol type is used to create unique identifiers for objects.

--> "typeof" operator:

-> Returns the type of the argument.

-> Syntax:

-> As an operator: typeof x. // same output

-> As a function: typeof(x). // same output

--> Type Conversions:

-> String conversion: let value = true; let var = String(value); // now var is a string

-> Numeric Conversion: let str = "123"; alert(typeof str); // string

--> Boolean Conversion:

-> Values that are intuitively "empty", like 0, an empty string, null, undefined, and NaN, become false.

-> Other values become true.

-> E.g: Boolean(0); // false, Boolean("0"); // true, Boolean("hello"); // true

--> Operators:

-> String concatenation / binary+ operator

-> E.g: let s = "my" + "string";

-> Numeric Conversion / unary+ operator

-> Assignment operator.

-> Remainder

-> Quotient

-> Exponential

-> Increment/decrement

-> BitWise operators (&, |, ^, ~, <<, >>, >>>)

-> Modify-in-place

-> Comma

--> Comparisons:

-> Boolean

-> String

-> Strict equality (===)

-> Comparison with null and undefined (null === undefined // false) (null == undefined // true)

--> Alert VS Prompt VS Confirm:

-> Alert:

-> This shows a message and pauses script execution until the user presses "OK".

-> E.g: alert("Hello");

-> Prompt:

-> The function prompt accepts two arguments.

-> E.g: prompt(title, [default]);

-> title = text to show

-> default = An optional second parameter, the initial value for the input field.

-> Confirm:

-> The function confirm shows a modal window with a question and two buttons: OK and Cancel.

-> E.g: confirm(question);

--> Conditional Operators:

-> "If"

-> "?"

--> Logical Operators:

-> OR, AND, NOT

--> Loops:

-> Types:

-> Entry controlled loop:

-> pre-checking loop

-> Exit controlled loop:

-> post-checking loop

-> while, do-while, for, foreach, forof

--> Switch Statement:

-> The switch has one or more case blocks and an optional default.

--> Fucntions:

-> Fucntion declaration

-> Local variable

-> Outer variable

-> Parameters

-> Default values

-> returning a value

--> Scopes:

-> Block Scope:

-> scope for variables declared using "let" and "const" not "var".

-> Like variable declared within "if" condition (not in function).

-> Function scope:

-> scope for variables declared using var, let and const.

-> scope with in the function.

-> Module scope:

-> Defined in a module (like component in angular) and uses in another module.

-> Need to immport module to use that module thing.

-> import './circle';

-> Scopes can be nested:

-> nested loopbs or if scopes.

-> Global scope:

-> top of file

-> Lexical scope:

-> Also known as "static scoping"

-> The inner function scope can access variables from the outer function scope.

-> The lexical scope consists of outer scopes determined statically.

-> Variables isolation:

-> You can reuse common variables names (count, index, current, value, etc) in different scopes without collisions.

--> Arrow Functions:

-> There's another very simple and concise syntax for creating functions, that's often better than Function Expressions.

-> E.g: let func = (arg1, arg2, ...argN) => expression

--> Debugging in chrome:

-> Sources Panel:

-> F12

-> Console

-> Breakpoints

-> Debugger command

-> pausing and look around

-> tracing the execution

-> Logging

--> Coding Style:

-> https://javascript.info/coding-style

-> Spaces

-> Curly braces position

-> Line length

-> Indents

-> Semicolons

-> Nesting Levels

-> Function Placement

--> Comments:

-> Use slashes for comments:

-> Use slashes for both single line and multi line comments.

-> Try to write comments that explain higher level mechanisms or clarify difficult segments of your code.

-> Don't use comments to restate trivial things.

-> Explanatory comments are usually bad.

-> Provide a high-level overview of components, how they interact, what's the control flow in various situations.

-> The bird's eye view of the code.

-> JSDoc = Special syntax for commenting.

-> JSDoc Tags = @author, @constructor, @deprecated, @params, @private, @return, @this

-> Types:

-> single line

-> multi-line

--> Ninja Code:

-> Code as short as possible.

-> One-letter variable.

-> Use abbreviations and understandable initials.

-> Reuse name (Add a new variable only when absolutely necessary).

-> Use Underscore or double underscore before variable name.

-> Overlap outer variable (Use same names for variables inside and outside a function. As simple. No efforts to invent new names).

--> Automated testing with Mocha:

-> Mocha is a feature-rich JavaScript test framework running on Node. js and in the browser, making asynchronous testing simple and fun.

-> Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

-> Mocha – the core framework: it provides common testing functions including describe and it and the main function that runs tests.

-> Chai – the library with many assertions.

-> Sinon – a library to spy over functions.

--> Garbage Collection:

-> Memory management in JavaScript is performed automatically and invisibly to us.

-> We create primitives, objects, functions… All that takes memory.

-> There's a background process in the JavaScript engine that is called garbage collector.

--> "This" keyword:

-> Keyword is used to point to the instance of an object from its own constructor and methods (when used inside function or class scope.)

-> It also keeps track of execution context also often referred to by some as the lexical scope or lexical environment.

-> E.g:

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

-> When used alone, the owner is the Global object, so this refers to the Global object.

-> E.g: var x = this;

--> Constructor Fucntions:

-> Constructor functions technically are regular functions. There are two conventions though:

-> They are named with capital letter first.

-> They should be executed only with "new" operator.

--> Primitive data types:

-> There are 8: boolean , byte , char , short , int , long , float and double .

--> Non-Primitive data types:

-> Strings, Arrays, Classes, Interface, etc.

--> Arrays:

-> let arr = new Array();

-> let arr = [];

-> Methods: Push/Pop (Stack), Shift(extracts an item from the beginning), UnShift(adds items to the beginning)

-> Splice: delete arr[1]; // remove first index element

-> Slice: arr.slice([start], [end]) // It returns a new array copying to it all items from index start to end - 1

-> Concat: arr.concat([3, 4]) // append 3,4 to arr array

--> Maps:

-> Map is a collection of keyed data items, just like an Object.

-> Collections of values.

-> A value in a Set may only occur once; it is unique in the Set's collection.

-> Can iterate its elements in insertion order.

-> functions:

```
-> let sayings = new Map();
-> sayings.set('dog', 'woof');
-> sayings.set('cat', 'meow');
-> sayings.set('elephant', 'toot');
-> sayings.size; // 3
-> sayings.get('fox'); // undefined
-> sayings.has('bird'); // false
-> sayings.delete('dog');
```

-> Example:

```
let wrongMap = new Map()
wrongMap['bla'] = 'blaa'
wrongMap['bla2'] = 'blaaa2'
console.log(wrongMap)  // Map { bla: 'blaa', bla2: 'blaaa2' }
```

--> Map function:

-> The map() method creates a new array with the results of calling a function for every array element.

-> The map() method calls the provided function once for each element in an array, in order.

-> map() does not execute the function for array elements without values.

-> this method does not change the original array.

-> Map calls a provided callback function once for each element in an array, in order, and constructs a new array from the results.

-> Callback is invoked only for indexes of the array which have assigned values (including undefined).

-> Example:

```
-> function myFunction(num) {
        return num * 10;
   }
   var numbers = [65, 44, 12, 4];
   var newarray = numbers.map(myFunction)
   console.log(newarray); // 650,440,120,40
```

-> Example:

```
const array1 = [1, 4, 9, 16];
const map1 = array1.map(x => x * 2); // pass a function to map
console.log(map1); // Output: Array [2, 8, 18, 32] // means array of these 4 elements
```

-> Not to use map if:

-> You're not using the array it returns (Actually map builds a new array).

-> You're not returning a value from the callback

--> Maps VS Objects:

-> Use maps over objects when keys are unknown until run time, and when all keys are the same type and all values are the same type.

-> Use maps if there is a need to store primitive values as keys because object treats each key as a string whether it's a number value, boolean value or any other primitive value.

-> Use objects when there is logic that operates on individual elements.

--> Maps VS Objects VS Arrays:

-> difference depends on the build-in functions of each of them.

--> Getting Unique Elements from array:

-> Sets:

-> Provided by ES6.

-> A collection of items which are unique i.e no element can be repeated.

-> Set can store any types of values whether primitive or objects.

-> Methods:

-> Set.prototype.add(), Set.prototype.delete(), Set.prototype.clear(), Set.prototype.entries(), Set.prototype.has(), Set.prototype.values(), Set.prototype.keys(), Set.prototype.forEach(), Set.protoype[@@iterator]()

-> Example:

```
-> var set1 = new Set(["sumit","sumit","amit","anil","anish"]);  // ["sumit","amit","anil","anish"]
-> var set2 = new Set("fooooooood"); // it contains 'f', 'o', 'd'
-> var set4 = new Set(); // it is an  empty set.
```

-> Example:

```
const ages = [26, 27, 26, 26, 28, 28, 29, 29, 30]
const uniqueAges = [...new Set(ages)]
```

-> Example:

```
const students = [
{
        name: 'Krunal',
        age: 26
},
{
        name: 'Ankit',
        age: 25
},
{
        name: 'Krunal',
        age: 26
}
]
const uniqueArr = [... new Set(students.map(data => data.name))]
```

-> Filter:

-> Example:

```
const ages = [26, 27, 26, 26, 28, 28, 29, 29, 30]
const uniqueAges = ages.filter((value, index, self) => self.indexOf(value) == index)
```

--> indexOf function:

-> indexOf() method returns a first index at which the given element can be found in an array, or -1 if it is not present in an array.

--> Date:

-> We can set out-of-range values, and it will auto-adjust itself.

-> Example:

-> let date = new Date(2013, 0, 32); // 32 Jan 2013 ?!?

-> alert(date); // ...is 1st Feb 2013!

-> Example:

-> // we have date1 and date2, which function faster returns their difference in ms?

```
function diffSubtract(date1, date2) {
return date2 - date1;
}

// or
function diffGetTime(date1, date2) {
return date2.getTime() - date1.getTime();
}
```

--> properties:

-> Types:

-> Data Properties:

-> Int, etc.

-> Accessor properties:

-> Getters, setters

-> We can build logic in setters and getters.

--> Classes:

-> In practice, we often need to create many objects of the same kind, like users, or goods or whatever.

-> In JavaScript, a class is a kind of function.

-> Classes always use strict.

-> Providing initial values for state.

-> Functions do specific things but classes are specific things.

-> For inheritance, use "extends" keyword.

-> Inheritance: new functionality on top of the existing.

-> Groups:

-> Internal interface – methods and properties, accessible from other methods of the class, but not from the outside.

-> External interface – methods and properties, accessible also from outside the class.

--> Mixins:

-> Like date mixins etc. // build in in loopback 3

-> If we want to do multiple inheritance which is not possible, we can do work similiar to it by using mixins concept.

-> Actually it is a class, containing methods that can be used by other classes without a need to inherit from it.

-> Provides methods that implement a certain behavior, but we do not use it alone, we use it to add the behavior to other classes.

--> Error Handling:

-> Uses where our script dies.

-> Types:

-> Try/Catch.

-> try and catch works synchronously.

-> If an exception happens in "scheduled" code, like in setTimeout, then try..catch won't catch it

-> Try/Catch/Finally

-> Custom errors, extending error.

-> Network error = HttpError = It includes Codes like 404, 403, 500, etc

-> Database error = DbError

-> Searching operatoins error = NotFoundError

-> Global Catch:

-> process.on("uncaughtException").

-> In the browser we can assign a function to the special window.onerror property, that will run in case of an uncaught error.

--> Async/Await, Promises, Callback

-> Callback:

-> "callback hell" or "pyramid of doom."

-> Promise:

-> Arguments: resolve(value), reject(error)

-> new Promise(function(resolve, reject) {});

-> promise.then(); promise.catch();

-> Basic Syntax:

-> var promise = doSomethingAync() , promise.then(onFulfilled, onRejected)

-> Q-Modules // This is also a promise library in node js

--> Generators & Iterators:

-> Standardize this process of looping over custom objects.

-> Special class of functions

-> Simplify the task of writing iterators.

-> Function that produces a sequence of results instead of a single value, i.e you generate a series of values.

-> Syntax: function generateSequence() {}*

-> Iterators are an implementation of Iterable objects such as maps, arrays and strings which enables us to iterate over them using next()

-> These are just like loops used to iterate the array but here array contains custom data format.

-> Main function of this is next()

-> This is suitable in complex scenarios like we have a object having an object with in. The nested object contains multiple arrays of string names. To get all the names, we need to use multiple loops in simple case. We can do it though iterable in a better way.

-> Expose all our internal data sequentially.

-> Standardize the process of looping over custom objects. Developer cannot return response in any format.

-> These are all iterables:

-> Arrays , Strings, Maps, Sets, etc

--> Spread Operators:

-> Copy and paste the elements at that position.

-> Sort of unpacking

-> Example:

```
const array = ['a','b'];
const newArray = [1, ...array, 2, 3];
console.log(newArray); // [1, 'a', 'b', 2, 3]
```

-> Example:

```
var a = [1,2,3]
funtion sp(a,b,c){
```

*return a*b*c;*

}

var b = sp(...a); //b = 6

-> Example:

var a = [1,2,3,4,5,6]

funtion sp(a,b,c){

*return a*b*c;*

}

var b = sp(...a); //b = 6

-> Example: (concating arrays)

var a= [1,2];

var b = [3,4];

var c = [...a, ...b]; then c =[1,2,3,4]

-> Math.max(...[-1, 5, 11, 3]) // dots in it are called spreading

-> Math.max()returns the largest one of its zero or more arguments.

-> Alas,it can't be used for Arrays, but spreading gives us a way out.

-> Math.max(-1, 5, 11, 3) -> 11

-> Math.max(...[-1, 5, 11, 3]) -> 11

--> Proxy:

-> It is one of the features that shipped with ES6.

-> It seems not to be widely used.

-> The Proxy object is used to define custom behavior for fundamental operations

(e.g. property lookup, assignment, enumeration, function invocation, etc).

-> In simple terms, proxies are getters and setters with lots of swag.

--> Eval:

-> The built-in eval function allows to execute a string of code.

-> E.g:

-> let code = 'alert("Hello")';

eval(code); // Hello

-> Can access outer local variables. That's considered bad practice.

--> Currying:

-> Advanced technique of working with functions.

-> Used in JavaScript as well as in other languages.

-> Transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c).

-> Currying doesn't call a function. It just transforms it.

-> Currying is a process in functional programming in which we can transform a function with multiple arguments into a sequence of nesting functions.

-> Lodash also provides this function as build-in.

-> Example:

```
function sum(a, b) {
        return a + b;
}

let curriedSum = _.curry(sum); // using _.curry from lodash library
alert( curriedSum(1, 2) ); // 3, still callable normally
```

```
                    alert( curriedSum(1)(2) ); // 3, called partially


          -> Example:
                    function multiply(a) {
                              return (b) => {
                                        return (c) => {
                                                  return a * b * c
                                        }
                              }
                    }
                    log(multiply(1)(2)(3)) // 6
```

--> Document:

-> Here's a bird's-eye view of what we have when JavaScript runs in a web-browser:

-> Window:

-> DOM: document, etc.

-> The Document Object Model (DOM) is a programming interface for HTML and XML documents.

-> JS HTML DOM.

-> With the HTML DOM, JavaScript can access and change all the elements of an HTML document.

-> Example:

```
<p id="demo"></p>
<script>
          document.getElementById("demo").innerHTML = "Hello World!"; // This will changes the content (the innerHTML) of the <p> element with id="demo"
</script>
```

-> BOM: navigator, screen, location, frames, history, XMLHttpRequest

-> Javascript: Object, Array, Function, etc.

-> Everything in HTML, even comments, becomes a part of the DOM.

-> When the browser loads the page, it "reads" (another word: "parses") the HTML and generates DOM objects from it.

-> E.g: For instance, if the tag is <body id="page">, then the DOM object has body.id="page".

-> HTML Doc -> Loads in browser -> Parsing(front-end js, tree structure called DOM).

-> Actually its an API through which developer can update HTML on run time.

--> Event Target:

-> Is the root "abstract" class.

-> Objects of that class are never created.

--> Browser Events:

-> Mouse events:

-> Click – when the mouse clicks on an element (touchscreen devices generate it on a tap).

-> Contextmenu – when the mouse right-clicks on an element.

-> Mouseover / mouseout – when the mouse cursor comes over / leaves an element.

-> Mousedown / mouseup – when the mouse button is pressed / released over an element.

-> Mousemove – when the mouse is moved.

-> Drag and Drop

-> Form element events:

-> Submit – when the visitor submits a <form>.

-> Focus – when the visitor focuses on an element, e.g. on an <input>.

-> Keyboard events:

-> keydown and keyup – when the visitor presses and then releases the button.

-> Document events:

-> DOMContentLoaded – when the HTML is loaded and processed, DOM is fully built.

-> CSS events:

-> Transitionend – when a CSS-animation finishes.

--> Bubbling:

-> When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

-> Handler: like onclick, etc.

-> Target: The most deeply nested element that caused the event is called a target element.

--> UI Events:

-> Mouse events (listed above)

-> Keyboard events (listed above)

-> Scrolling

--> Form Control Events:

-> Change

-> Input

-> Cut

-> Copy

-> Paste

--> IFrame:

-> An HTML document embedded inside another HTML document on a website.

-> The IFrame HTML element is often used to insert content from another source, such as an advertisement, into a Web page

--> Resource Loading:

-> The browser allows us to track the loading of external resources – scripts, iframes(Inline Frame), pictures and so on.

-> There are two events for it:

-> onload – successful load

-> onerror – an error occurred.

--> Cross-window communication:

-> The "Same Origin" (same site) policy limits access of windows and frames to each other.

-> So, the purpose of the "Same Origin" policy is to protect users from information theft.

-> E.g: // Below all have same origin

-> http://site.com

-> http://site.com/

-> http://site.com/my/page.html

--> The clickjacking attack:

-> The "clickjacking" attack allows an evil page to click on a "victim site" on behalf of the visitor.

-> Many sites were hacked this way, including Twitter, Facebook, Paypal and other sites. They have all been fixed, of course.

-> Here's how clickjacking was done with Facebook:

-> A visitor is lured to the evil page. It doesn't matter how.

-> The page has a harmless-looking link on it (like "get rich now" or "click here, very funny").

-> Over that link the evil page positions a transparent <iframe> with src from facebook.com, in such a way that the "Like" button is right above that link. Usually that's done with z-index.

-> In attempting to click the link, the visitor in fact clicks the button.

--> Chrome extensions:

-> "Google Chrome Extensions are small software programs that can modify and enhance the functionality of the Chrome browser".

-> To develop a Google Chrome Extension you should write some javascript and or html/css. Then you can run the extension in your browser.

-> If you wish for others to download your extension you will have to provide config.json file that describes you extension sets permissions etc.

-> A group of files that customizes your experience in the Google Chrome browser.

--> Electron:

-> Framework for creating native applications with web technologies like JavaScript, HTML, and CSS.

-> It takes care of the hard parts so you can focus on the core of your application.

-> Developer: GitHub

->Languages used: JavaScript

--> ES6 VS ES5:

-> ECMAScript 6 is also known as ES6 and ECMAScript 2015.

-> ES6 features: let, const, JS arrow function, JS classes, default parameter values, array.find(),

array.findIndex(), exponentil (**), Number.isInteger()Number.isSafeInteger(), isFinite(), isFinite(), map,

promises, proxies

-> ES6 Supported browsers: chrome, explorer, firefox, safari, opera

*-> ES5: var x = function(x, y) { return x * y; }*

*-> ES6: const x = (x, y) => x * y;*

-> ES6 introduced classes.

-> ES6 Classes:

-> A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class

-> The properties are assigned inside a constructor() method.

-> Use the keyword class to create a class, and always add a constructor method.

-> A lot of communities are supporting ES6.

--> VanillaJS:

-> It is plain javascript.

-> A term for library/framework free javascript.

-> Powerful web applications without jQuery.

-> Var VS Let:

-> var is old and let/const is new.

-> var has no block scope.

-> var declarations are globally scoped or function scoped while let and const are block scoped.

-> While var and let can be declared without being initialized, const must be initialized during declaration.

-> When you use var, you can declare the same variable as many times as you like, but with let you can't.

-> Global scope => Block scope access or Lexical scope.

-> In strict mode, var allows a variable to be re-declared in the same scope => Do not allow.

->

--> Coupling and Cohesion:

-> Cohesion: sticking together, Measure of how well module fits together.

-> Code should contains low coupling(modules should be independent as mush as possible)

-> Low coupling is for easiness in changing one module without affecting the other one.

-> Coupling: An indication of the strength of interconnections between program units.

--> Loops: (NOTE: I still have not confirmed the below async and sync of loops)

-> while // Asyncronous (non-blocking)

-> do-while // Asyncronous

-> for // synchronous

-> for-of //

-> for-await-of // like for-of.

-> for-in // has several pitfalls. Therefore,it is not recomended.

-> for-each // Synchoronous (blocking) (executes the provided callback once for each element present in the array in ascending order.)

--> Exceptions:

-> • try-catch • try-finally • try-catch-finally

-> finally block is the block that will always execute

--> Built-in functions:

-> .equals, .isFinite, .isInteger, .isNaN, .isSafeInteger, .parseInt, .parseFloat, .map,
Math.floor, Math.ceil, Math.round, Math.trunc, Math.max, eval, etc

--> Static methods of Number:

-> .isFinite, .isInteger, .isNaN, .isSafeInteger, .parseInt, .parseFloat

--> Undefined VS Null:

-> undefined is the non-value used by the language(when something is uninitialized, etc.)

-> null means "explicitly switched off".

--> Equality:

-> loose equality (==)

-> Strict equality (===) : NaN === NaN gives true

-> Object.is(2, 2): most strict, Object.is(NaN, NaN) gives false.

--> Primitive/atomic values VS object/compound values:

-> Primitive: undefined, null, boolean, number, string, symbol

-> Object: Array, map, set, function, date, RegExp(regular expression)

--> Strict mode vs sloppy mode:

-> Normal"sloppy"mode is the default in scripts.

-> Strict mode is the default mode in modules and trhe classes.

-> 'use strict';

--> typeOf:

-> undefined 'undefined', null 'object' , etc.

--> Normal Equality VS Deep Equality:

->

--> Exception Handling:

-> throws

-> try catch

--> Statement VS Expression:

-> A statement is a Piece of code that can be executed and performs some actions.

-> An expression is a peice of code that can be evaluated to produce a value.

--> FCM:

-> await Productcategory.destroyAll({productCategoryId:{inq:[2,3]}}); // no need of where keyword.

-> FCM = Firebase Cloud Messaging.new version of Google Cloud Messaging(GLM).

-> For use cases such as instant messaging, a message can transfer a payload of up to 4KB to a client app.

--> console build-in functions/APIs: (almost 20+ overall)

-> console.clear();// It clears the screen

-> console.table();// Displays tabular data as a table

-> console.timeEnd(); // Starts a timer with a name specified as an input parameter. Up to 10,000 simultaneous timers can run on a given page.

-> console.timeEnd();// Stops the specified timer and logs the elapsed time in seconds since it started.

-> console.timeLog(); // Logs the value of the specified timer to the console.

--> setTimeout():

-> Execute a specified block of code once after a specified time has elapsed.

--> setInterval():

-> Execute a specified block of code repeatedly with a fixed time delay between each call.

--> Add JS to your code:

-> <script> internal js code here </script>

-> <script src="script.js" defer></script> // import file having js

--> Split:

-> returns an array by spliting the string.

-> Example: Convert string to array

```
-> let myData = 'Manchester,London,Liverpool,Birmingham,Leeds,Carlisle';
       let myArray = myData.split(',');
```

--> Join:

-> Opposite to split.

-> let myNewString = myArray.join(',');

--> Events in node js:

-> Every action on a computer is an event. Like when a connection is made or a file is opened.

--> Node Lifecycle:

-> node App.js => Start executing script => Code parsing, Register events and functions => event loop=> keeps on running as long as event are registered.

--> Event Loop in node js:

-> Node.js is a single-threaded application, but it can support concurrency via the concept of event and callbacks.

-> Every API of Node.js is asynchronous and being single-threaded, they use async function calls to maintain concurrency

-> As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.

-> In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.

-> Event Emitters => Events (list/Queue) => Event Loop => Event Handler

-> Application (Gives tasks/callbacks to event loop, receives cb from event loop) => Event Loop(Single thread, Non-Blocking Operations, creates threads, take return from threads, return cb to the application) => Threads(Blocking Operations)

-> The functions that listen to events act as Observers.

-> Whenever an event gets fired, its listener function starts executing.

-> Node.js uses a non-blocking I/O model and asynchronous programming style.

-> Is Node.js single-threaded?

-> Yes! And you are right.

-> No! And You are right again.

-> Also people use many expressions like multitasking, single-threaded, multi-threaded, thread pool, epoll loop, event loop, and so on.

-> When there is a single-core processor, and the processor can process only one task at a time, application calls yield after it has finished, to notify the processor to start process the next task

-> Thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.

->

--> Array Destructuring:

-> Examples:

-> simple case:

```
var a = [1,2,3,4];
var [x,y] = a; //x =1, y = 2
```

-> simple but with default values:

```
var b = [1];
var [x=0,y=3] = b; //x= 1,y=3
```

-> swaping two variables:

```
var q = 1;
var w = 2;
[w,q] = [q,w]; //man it's just like python
```

-> extracting one and rest as array:

```
var a =[1,2,3,4]
var [ one, ...rest ]  = a; // one = 1 rest = [2,3,4]
```

--> Object Destructuring:

-> extracting the property values from an object

-> Example:

-> simple:

var o = {'p' : 1,'q' : 7};

var { p,q } = o; //p : 1, q :7

-> new variablenames:

var o = {'p' : 1, 'q' : 5};

var { p:a, q:b } = o ;//declares variables a=1, b = 5

--> Arrow Functions:

-> Es6

-> Example:

```
func = ((a,b,c) => {
return a+b+c;
});
console.log(b(1,2,3)); // 6
```

--> Filter VS find:

-> The find() method returns the first value that matches from the collection

-> Once it matches the value in findings, it will not check the remaining values in the array collection.

-> The filter() method returns the matched values in an array from the collection.

--> Briefing of projects till now in Express JS:

-> Notes:

-> compiling starts from app.js / server.js

-> App.js:

-> require('file path') = requires a file of the same project.

-> app.use(cors()); = use to allow the user for cross-domain requests.

-> app.use(bodyParser.json()); = use for parsing. use this line on top of file. Parses the text as JSON and exposes the resulting object on req.body.

-> bodyParser.text(): Reads the buffer as plain text and exposes the resulting string on req.body.

-> bodyParser.urlencoded({ extended: false }) = bodyParser.urlencoded

-> Middlewares:

-> body-parsers etc are middlewares.

-> unctions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle.

-> The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

-> Routes folder:

-> create js file against each model.

-> APIs will be written here with their routes.

->

-> ORM:

-> Sequelize

--> Briefing of projects till now in Loopback 3:

-> Notes:

-> logger used = loopback-component-winston

->

-> Templates:

-> ejs files.

-> Like email sending template

-> Folders:

-> Common => models

-> Server

-> boot

-> explorer

-> middleware (url-not-found.js etc)

-> server.js

-> middlware.json

-> datasources.json

->

--> Speed up the loops:

https://www.aivosto.com/articles/loopopt.html#loops

-> performance bottlenecks of an application.

-> The key to speed up the program is to make the loops run faster.

-> With nested loops, put your efforts on the innermost loop. It's the one that executes the largest number of times.

-> If the loop should execute at least once, use a post-condition loop. What's the deal? You save one evaluation.

-> Use pre-calculated values (example below).

-> Techniques:

1. Use lookup tables.

2. Use less exponential operator ^

-> Example:

```
For y = 0 to Height-1
For x = 0 to Width-1
          i = y*Width + x -- here y*Width is invariant or constant. so use it in upper loop despite inner.
          Process i
Next x
Next y

-- Loop after optimization
For y = 0 to Height-1
     Temp = y*Width
     For x = 0 to Width-1
          i = Temp + x
          Process i
     Next x
Next y
```

3. Expand loops:

Example (loop2 is faster than loop1):

```
-- loop1
For y = 0 to 7
     i(y) = z And 2^y
Next
```

```
-- loop2
For y = 0 to 7
    i(y) = j(0) And 2^y + j(1) And 2^(y-1) + j(2) And 2^(y-2) + j(3) And 2^(y-3)
Next
```

4. Eliminate temporary storage variables.

5. Choose faster operators:

x^2	*is slower that x*x.*
x Mod 16	*is slower than x And 15.*
x / y	*is slower than x \ y (Integer division can run faster than floating point division).*
*x*x*x*x as temp=x*x*	*is slower than result=temp*temp.*

6. Optimize the control flow:

1.If-Else-Then:

```
-- First If-else
If condition Then
  rare_block
Else
  usual_block
End If

-- Second If-Else
If Not condition Then
  usual_block
Else
  rare_block
End If
```

7. Short-circuit conditionals:

-- 1 is slower than 2 from below.

1. Evaluate both x and y -> If both are True, return True -> Otherwise return False

2. Evaluate x -> If x is False, return False -> Otherwise evaluate y -> If y is False, return False, otherwise return True

8. Remove recursion:

You remove recursion by replacing it with a loop.

Sometimes it is enough to use GoTo to jump to the start of the function.

Sometimes you need to do more, like arrange parameter values and local variables.

--> QUERIES:

-> to how many levels we can use nested callbacks in node js ?

--> Other Guidelines:

-> Code should be shallow.

-> You can also use Promises, Generators and Async functions to fix callback hell(pyramid shape).

--> Middlewares:

-> An Express application can use the following types of middleware:

-> Application-level middleware

-> Router-level middleware

-> Error-handling middleware

-> Built-in middleware

-> Third-party middleware

--> Why should we use Node JS:

-> Node.js is primarily used for non-blocking (Not for blocking), event-driven servers, due to its single-threaded nature.

-> Handle a large amount of simultaneous connections in a non-blocking manner.

-> It's especially useful for proxying different services with different response times, or collecting data from multiple source points.

-> Server-side application communicating with third-party resources, pulling in data from different sources, or storing assets like images and videos to third-party cloud services.

-> Pros:

-> If your application doesn't have any CPU intensive computation, you can build it in Javascript top-to-bottom, even down to the database level if you use JSON storage Object DB like MongoDB.

-> Cons:

-> Any CPU intensive computation will block Node.js responsiveness.

-> Using Node.js with a relational database is still quite a pain.

-> Where Node.js Shouldn't Be Used:

-> SERVER-SIDE WEB APPLICATION WITH A RELATIONAL DB BEHIND.

-> HEAVY SERVER-SIDE COMPUTATION/PROCESSING or Fibonacci computation.

--> Is Node.js a framework?

-> No, it's an environment, and back-end frameworks run within it.

-> Popular ones include Express.js (or simply Express) for HTTP servers and Socket.IO for WebSocket servers.

--> Is Node.js a programming language?

-> No, the ".js" means that the programming language you use with Node.js is JavaScript (or anything that can transpile to it, like TypeScript, Haxe, or CoffeeScript.)

--> What is the difference between Node.js and Angular/AngularJS?

-> Node.js executes JavaScript code in its environment on the server.

-> Angular is a JavaScript framework that gets executed on the client (i.e. within a web browser.)

--> Javascript VS NodeJS:

-> JavaScript is a simple programming language which runs in any browser JavaScript Engine.

Whereas Node JS is an interpreter or running environment for a JavaScript programming language which holds

a lot of excesses require libraries which can easily be accessed from JavaScript programming for better use.

-> Node js runs javascript code.

-> Node JS is a server-side JS engine.

-> Normally all browsers have JavaScript engine that helps us to run javascript in a web browser.

-> Spider monkey (FireFox), JavaScript Core (Safari), V8 (Google Chrome) are some popular javascript engine using verities browsers.

-> But node js is using V8 engine directly, with some libraries to do some I/O or networking operations.

-> Node JS is some extension of JavaScript libraries.

-> Node JS manages security like tracing MAC IP.

-> Javascript for Server-side was unimaginable before Node.js.

-> Without Node JS:

-> Either depend on another developer who can/should do backend/server-side programming for them.

-> Or learn a new server-side programming language.

-> Type:

Programming Language => interpreter and environment fo js with some libraries.

-> Utility:

mainly usage for clinet-side like validations etc => mainly usage for performing any non-blocking operations of OS.

-> Running Engine:

Running in any engine like spider monkey(Fire Fox) etc => Uses V8 engine

-> Usage:

-> Designed to create network centric applications as it is integrated with HTML and Java => Used for data intensive real time based applications.

-> Dedicated server:

-> No => Yes

-> Companies Uses these:

-> Uber, Shopify, Slack, etc => Netflix, Linkedin, Uber, Paypal, eBay, Medium, NASA, NEw York Times, etc.

-> Limitations:

-> Doesn't allow R/W of file, no multi threading, no multi processor, interpreted differently by different browsers => Single threaded, Not suitable for heavy computational apps

--> To run angular project on 3000:

-> ng build --prod / ng build

-> copy files and place in to client folder.

-> give client folder path in the middleware.json

```
->   "files": {
                    "loopback#static": {
                    "params": "$!../client"
                    }
          }
```

-> comment or del root.js in boot folder.

--> Empty-String VS Null VS Undefined:

-> Null is an assigned value. It means nothing.

-> Undefined typically means a variable has been declared but not defined yet.

-> Empty string is a string instance of zero length.

--> Hashing & Salt:

-> Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string.

-> Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value.

-> Hashing provides constant time search, insert and delete operations on average.

-> Hashing and Salt are techniques to store sensitive data such as passwords securely.

-> Hashing is the irreversible conversion of strings.

-> Hash function will convert arbitrary length strings into fixed length strings.

-> If you apply a hash function to the same strings, it will return the same values.

-> Salt adds arbitrary strings before applying hash functions, which will make hashed string more unpredictable.

-> Example:

-> Hashing: Crypto library for password etc encryption.

--> Hash Table / Lookup Table:

-> Data structure.

-> Used to store keys/value pairs.

-> It uses a hash function to compute an index into an array in which an element will be inserted or searched.

-> O(1)

-> Uses when data is big.

-> Store most frequent data.

-> Example:

-> Let suppose we have a string "ababcd" and we want to calculate the frequency of all the letters in this string.

means occurence of a,b,c,d,e, ..., z.

-> Without Hashing:

*-> Complexity: O(26*N) where N=size of string, 26=total letters.*

*-> We will use nested loop for iteration having outerloop with length of string and inner with 26 (N*26);*

-> With Hashing:

-> Take an array of size=26 anf fill all letters in it.

-> Iterate with size of string and increment the frequencyCount against each letter index.

-> Then iterate a loop with size=26 to display the counts.

-> O(N+N) = O(N)

--> Lexical scoping VS Function scope VS Global scope:

-> Lexical Scope:

-> child resource can access parent scope resuorces like inner loop using outer loop variables.

-> Function Scope:

-> Variables with in curly braces.

-> if they are declared with "var", they will be accessable in whole program WHILE THE FUNCTION IS RUNNING.

-> Global Scope:

-> variable will long last till the whole program executes.

--> Sorting:

-> array_name.sort() // in ascending order

--> Express Modules:

-> Authentication:

-> Modules:

-> Hash and salt.

-> Sessions or JWT.

-> Cookie-Session VS JWT:

-> Intro:

-> HTTP is a stateless protocol, to overcome this, we can use either session or tokens.

-> Example: To keep state in shoping cart system.

-> The cookie created above is a session cookie: it is deleted when the client shuts down, because it didn't specify an Expires or Max-Age directive.

-> Session ID created which is stored in a cookie in the client's browser while the user performs certain activity on the website.

-> On every request that the user makes, a cookie is sent along with it.

-> Cookie-Session:

-> A session cookie contains information that is stored in a temporary memory location and then subsequently deleted after the session is completed or the web browser is closed.

-> A session cookie is also known as transient cookie.

-> Normal Sessions are stored on server side as well as on client side.

-> Cookie Sessions (Session Cookie or Cookie) are stored on client side.

-> JWT:

-> JSON Web Tokens.

-> Modern applications uses it.

-> This is the widely used method for RESTful APIs.

-> Token can be stored in db (better), client side.

-> A JWT needs to be stored in a safe place inside the user's browser.

-> Don't store it in local storage (or session storage).

-> If you store it inside localStorage, it's accessible by any script inside your page (which is as bad as it sounds, as an XSS attack can let an external attacker get access to the token).

-> The JWT is stored on the client side usually in localStorage (bad apprach) and it is sent as an unique key of that user when the user requests any data from the server or is performing any activity for that website.

-> When the request is received by the server, it validates the JWT for every request that it is that particular user only and then sends the required response back to the client.

-> Parts (concatenated by dots '.'):

-> Header (Base64-URL string):

-> Example:

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

-> Payload (Base64-URL string):

-> Example:

```
{
    "sub": "1234567890",
    "name": "John doe",
    "admin": true
}
```

-> Signature (Base64-URL string)

-> Cookies:

-> An HTTP cookie is a small piece of data stored on the user's computer by the web browser while browsing a website

-> Cookies were designed to be a reliable mechanism for websites to remember stateful information or to record the user's browsing activity

-> MarkDown:

-> md

-> Template Engine

-> Multi Router:

-> If we have same route with different logic in APIs versions, we use multiple routes.

-> We create multiple APIs versions folders.

-> Example:

-> api/v1/get-all-users

-> api/v2/get-all-users

-> Example:

-> Current URL for login of gmail using v2 is:

-> Email screen: https://accounts.google.com/signin/v2/identifier?continue

-> Password screen: https://accounts.google.com/signin/v2/identifier?continue=

-> MVC:

->

->

-> Online:

-> Tracking online user activity with online and redis packages

-> Route Seperation:

-> Creates routes file for each model like user, etc.

-> Middlewares:

-> express.static()

--> Function Parameters sequence:

-> Err:

-> First Place.

-> It should be on first place beacause Errors that occur within asynchronous callbacks are easy to miss.

-> Therefore, as a general principle first argument to the asynchronous calls should be an Error object.

--> NodeJS Security Enhancement Techniques:

-> Application Security:

-> Process of developing, adding, and testing security features within applications to prevent security vulnerabilities against threats such as unauthorized access and modification.

-> Use flat Promise chains (Promises):

-> Asynchronous callback functions are one of the strongest features of Node.js.

-> Solution of Pyramid of Doom or Callback Hell.

-> Another advantage of Promises is the way Promises handle the errors.

-> Uses .then and .catch for errror handling.

-> Set request size limits:

-> Buffering and parsing of request bodies can be resource intensive for the server.

-> If there is no limit on the size of requests, attackers can send request with large request bodies so that they can exhaust server memory or fill disk space.

-> However, fixing a request size limit for all requests may not be the correct behavior, since some requests like those for uploading a file to the server have more content to carry on the request body.

-> Example:

-> app.use(express.urlencoded({ limit: "1kb" }));

-> app.use(express.json({ limit: "1kb" }));

-> app.use(express.multipart({ limit:"10mb" }));

-> app.use(express.limit("5kb")); // this will be valid for every other content type

-> Do not block the event loop:

-> Node.js is very different from common application platforms that use threads.

-> Node.js has a single-thread event-driven architecture.

-> By means of this architecture, throughput becomes high and programming model becomes simpler.

-> The event loop looks for events and dispatches them to handler functions.

Because of this, when CPU intensive JavaScript operations are done, the event loop waits for them to finish.

This is why such operations are called blocking.

-> To overcome this problem, Node.js allows assigning callbacks to IO-blocked events.

-> This way, the main application is not blocked and callbacks run asynchronously.

-> Therefore, as a general principle, all blocking operations should be done asynchronously so that the event loop is not blocked.

-> Even if you perform blocking operations asynchronously, it is still possible that your application may not serve as expected.

This happens if there is a code outside the callback which relies on the code within the callback to run first.

-> Perform input validation:

-> Input validation is a crucial part of application security.

-> Perform output escaping (encoding):

-> A technique used to ensure that characters are treated as data,

not as characters that are relevant to the interpreter's parser.

-> This means that someone can create a URL like

http://example.com/yourscript.php?name=<script src="http://evil.site/evil.js"></script> and then then send that URL to someone else.

-> The escape() function computes a new string in which certain characters have been replaced by a hexadecimal escape sequence.

-> In addition to input validation, you should escape all HTML and JavaScript content shown to users via

application in order to prevent cross-site scripting (XSS) attacks.

-> You can use escape-html or node-esapi libraries to perform output escaping.

-> Example:

-> escape('ć'); // "%u0107"

-> Perform application activity logging:

-> Logging application activity is an encouraged good practice.

-> It makes it easier to debug any errors encountered during application runtime.

-> It is also useful for security concerns, since it can be used during incident response.

-> Example:

-> Winston.

-> Bunyan.

-> Monitor the event loop:

-> When your application server is under heavy network traffic, it may not be able to serve its users.

-> This is essentially a type of Denial of Service (DoS) attack.

-> The "toobusy-js" module allows you to monitor the event loop.

-> It keeps track of the response time, and when it goes beyond a certain threshold, this module can indicate your server is too busy.

-> In that case, you can stop processing incoming requests and send them 503

Server Too Busy message so that your application stay responsive.

-> Example:

```
->  var toobusy = require('toobusy-js');
          app.use(function(req, res, next) {
                    if (toobusy()) {
                              // log if you see necessary
                              res.send(503, "Server Too Busy");
                    } else {
                    next();
                    }
          });
```

-> Take precautions against brute-forcing:

-> Attackers can use brute-forcing as a password guessing attack to obtain account passwords.

-> Node.js has several modules available for this purpose.

-> Example:

-> Express-bouncer, express-brute and rate-limiter are just some examples of the libraries.

-> Express-bouncer and express-brute modules work very similar and they both increase the delay with each failed request.

-> They can both be arranged for a specific route.

-> CAPTCHA usage is also another common mechanism used against brute-forcing.

-> Example:

-> "svg-captcha" library.

-> Use Anti-CSRF tokens:

-> It aims to perform authorized actions on behalf of an authenticated user, while the user is unaware of this action.

-> CSRF attacks are generally performed for state-changing requests like changing a password, adding users or placing orders.

-> "Csurf" is an express middleware that can be used to mitigate CSRF attacks.

-> Remove unnecessary routes:

-> A web application should not contain any page that is not used by users, as it may increase the attack surface of the application.

-> This occurs especially in frameworks like Sails and Feathers, as they automatically generate REST API endpoints.

-> Example:

-> In Sails, if a URL does not match a custom route, it may match one of the automatic routes and still generate a response.

-> Prevent HTTP Parameter Pollution (HTTP PP):

-> An attack in which attackers send multiple HTTP parameters with the same name and this causes your application to interpret them in an unpredictable way.

-> When multiple parameter values are sent, Express populates them in an array.

-> In order to solve this issue, you can use "hpp" module.

-> Example:

```
-> var hpp = require('hpp');
   app.use(hpp());
```

-> Only return what is necessary:

-> Return only the required data despite the complete json object.

-> Use object property descriptors:

-> It include 3 hidden attributes:

-> writable (if false, property value cannot be changed)

-> enumerable (if false, property cannot be used in for loops)

-> configurable (if false, property cannot be deleted)

-> When defining an object property through assignment, these three hidden attributes are set to true by default.

-> Use access control lists:

-> Authorization prevents users from acting outside of their intended permissions.

-> Example:

-> ACL module.

-> Error & Exception Handling:

-> Handle uncaughtException:

-> Node.js default behavior for uncaught exceptions is to print current stack trace and then terminate the thread.

-> Node.js allows customization of this behavior.

-> The correct use of ' uncaughtException ' is to perform synchronous cleanup of allocated resources (e.g. file descriptors, handles, etc) before shutting down the process.

-> The 'uncaughtException' event is emitted when an uncaught JavaScript exception bubbles all the way back to the event loop.

-> Example:

```
-> fs.readFile('somefile.txt', function (err, data) { //.. code here ..// }
```

If and error occurs while reading, it will be caught in ".catch".

But of there is no file exist, it will cause error and stops the working of the app (crashes the app).

-> Syntax:

```
-> process.on("uncaughtException", function(err) {
          // clean up allocated resources
          // log necessary error details to log files
          process.exit(); // exit the process to avoid unknown state
```

});

-> Listen to errors when using EventEmitter.

-> Handle errors in asynchronous calls.

-> Server Security:

-> Set cookie flags appropriately:

-> Generally, session information is sent using cookies in web applications.

-> However, improper use of HTTP cookies can render an application to several session management vulnerabilities.

-> There are some flags that can be set for each cookie to prevent these kinds of attacks.

-> "httpOnly" flag prevents the cookie from being accessed by client-side JavaScript.

-> "Secure" flag lets the cookie to be sent only if the communication is over HTTPS.

-> Use appropriate security headers:

-> Strict-Transport-Security:

-> X-Frame-Options:

-> X-XSS-Protection:

-> X-Content-Type-Options:

-> Content-Security-Policy:

-> Cache-Control and Pragma:

-> X-Download-Options:

-> Expect-CT:

-> Public-Key-Pins:

-> X-Powered-By:

-> Platform Security:

-> Keep your packages up-to-date:

-> Security of your application depends directly on how secure the third-party packages you use in your application are.

-> Also you can use Retire.js to check JavaScript libraries with known vulnerabilities.

-> Do not use dangerous functions:

-> Example:

-> eval()

-> Stay away from evil regexes.

-> Run security linters periodically:

-> When developing code, keeping all security tips in mind can be really difficult.

-> This is why there are Static Analysis Security Testing (SAST) tools.

-> These tools do not execute your code, but they simply look for patterns that can contain security risks.

-> As JavaScript is a dynamic and loosely-typed language, linting tools are really essential in the software development life cycle.

-> These tools should be run periodically and the findings should be audited.

-> Example:

-> ESLint

-> JSHint

-> Use strict mode:

-> JavaScript has a number of unsafe and dangerous legacy features that should not be used.

In order to remove these features, ES5 included a strict mode for developers.

-> It also helps JavaScript engines perform optimizations.

-> Adhere to general application security principles:

--> Access Control List (ACL):

-> https://blog.nodeswat.com/implement-access-control-in-node-js-8567e7b484d1

->

->

--> Package Manager Usage:

-> NPM: 60%

-> YARN: 13%

-> Bower: <1%

-> Duo: <1%

--> Node.js alternatives (But not even close to Node js):

-> Rhino

-> Nashron

--> JavaScript Apply VS Call:

-> Definition:

-> Used to borrow functions and to set this value => Derived to borrow and invoke functions

-> Takes arguments as an array => Takes arguments separately

-> Example:

```
-> var person = {
            fullName: function() {
                  return this.firstName + " " + this.lastName;
            }
      }
      person.fullName.apply(person1);  // Will return "Mary Doe"
```

-> Built-in function:

-> Built-in function like min and max functions => Cannot be used.

--> JavaScript vs JScript:

-> Both are scripting lang.

--> JavaScript vs Ruby:

-> OOPS:

-> Both are object-oriented scripting language =>

-> Programming Language :

-> Front-end programming language, mainly used for client-side application, server-side, browser level interaction and changes => Back-end programming language.

-> Performance:

-> More than 20 times faster than Ruby in certain cases due to its highly optimized engine.

-> When to Use:

-> Can be used if your application needs to develop in less time, performance and scalability => Applications which are CPU heavy applications

--> Require vs Import:

->

->

--> Security Tools:

-> NodeSecurity.io:

->

->

-> Sync.io:

->

->

-> Automatically restart production server:

-> Strong loop process manager:

->

-> PM TWO:

->

-> Forever and Forever Monitor:

->

-> OKAY Library:

-> To avoid callback hell.

->

--> Disadvantages of JavaScript:

-> No support for multithreading

-> No support for multiprocessing

-> Reading and writing of files is not allowed

-> No support for networking applications.

--> Types of Functions:

-> Named:

-> function display() {}

-> Anonymous:

-> var display=function() {}

--> Closure:

-> We need closures when a variable which is defined outside the scope in reference is accessed from some inner scope.

-> Example:

```
-> var num = 10;
   function sum()
   {
   num++;
   }
```

--> JavaScript VS JScript:

-> Netscape provided the JavaScript language. Microsoft changed the name and called it JScript to avoid the trademark issue.

-> In other words, you can say JScript is the same as JavaScript, but Microsoft provides it.

--> BOM:

-> Browser Object Model.

-> It provides interaction with the browser.

-> The default object of a browser is a window.

-> The window object provides various properties like document, history, screen, navigator, location,

innerHeight, innerWidth, alert, confirm, promt, open, close, setTimeout, etc.

-> Example:

-> window.location.reload();

--> DOM:

-> Document Object Model.

-> A document object represents the HTML document.

-> It can be used to access and change the content of HTML.

--> History object of browser:

-> The history object of a browser can be used to switch to history pages such as back and forward from the current page or another page.

-> There are three methods of history object:

-> history.back() - It loads the previous page.

-> history.forward() - It loads the next page.

-> history.go(number) - The number may be positive for forward, negative for backward. It loads the given page number.

--> Data Types:

-> Primitive:

-> String, Number, Boolean, Undefined, Null

-> Non-primitive:

-> Object, Array, RegExp

--> HTML code dynamically using JavaScript:

-> Using innerHTML tag

-> Example:

-> document.getElementById('mylocation').innerHTML="<h2>This is heading</h2>";

--> Object creation:

-> There are 3 ways to create an object in JavaScript.

-> By object literal (Example: emp={id:102,name:"Rahul Kumar",salary:50000})

-> By creating an instance of Object

-> By Object Constructor

-> Array creation:

-> By array literal (Example: var emp=["Shyam","Vimal","Ratan"];)

-> By creating an instance of Array

-> By using an Array constructor

--> isNaN function:

-> The isNan() function returns true if the variable value is not a number.

--> Client side JavaScript VS Server side JavaScript:

-> Client side:

-> Basic language and predefined objects which are relevant to running JavaScript in a browser.

-> The client-side JavaScript is embedded directly by in the HTML pages.

-> he browser interprets this script at runtime.

-> Server side:

-> Resembles client-side JavaScript.

-> It has a relevant JavaScript which is to run in a server.

-> The server-side JavaScript are deployed only after compilation.

--> In which location cookies are stored on the hard disk:

-> The storage of cookies on the hard disk depends on the OS and the browser.

-> The Netscape Navigator on Windows uses a cookies.txt file that contains all the cookies.

-> The path is c:\Program Files\Netscape\Users\username\cookies.txt

-> The Internet Explorer stores the cookies on a file username@website.txt.

-> The path is: c:\Windows\Cookies\username@Website.txt.

--> What is this [[[]]]?

-> Three dimentional array.

--> Pop-up boxes:

-> Alert, Confirm, Prompt.

--> Is JavaScript faster than ASP script?

-> Yes, because it doesn't require web server's support for execution.

--> How to validate email in JavaScript?

-> Example:

```
var x=document.myform.email.value;
var atposition=x.indexOf("@");
var dotposition=x.lastIndexOf(".");
if (atposition<1 || dotposition<atposition+2 || dotposition+2>=x.length){  }
```

--> What is this keyword in JavaScript?

-> The this keyword is a reference variable that refers to the current object.

--> Requirement of debugging in JavaScript:

-> Using console.log() method

-> Using debugger keyword

--> Node.js web application architecture:

-> Client Layer:

-> The Client layer contains web browsers, mobile browsers or applications which can make an HTTP request to the web server.

-> Server Layer:

-> The Server layer contains the Web server which can intercept the request made by clients and pass them the response.

-> Business Layer:

-> The business layer contains application server which is utilized by the web server to do required processing.

-> This layer interacts with the data layer via database or some external programs.

-> Data Layer:

-> The Data layer contains databases or any source of data.

--> Types of API functions:

-> Asynchronous, Non-blocking functions

-> Synchronous, Blocking functions

--> Error-first callback:

-> Error-first callbacks are used to pass errors and data.

-> If something goes wrong, the programmer has to check the first argument because it is always an error argument.

-> Additional arguments are used to pass data.

-> Example:

-> fs.readFile(filePath, function(err, data) {}

--> Built-in Debugger:

-> Node.js provides a simple TCP based protocol and built-in debugging client.

--> Control Flow functions:

-> Control flow function is a generic piece of code that runs in between several asynchronous function calls.

-> The control flow does the following job:

-> Control the order of execution

-> Collect data

-> Limit concurrency

-> Call the next step in a program

--> Is it possible to access DOM in Node:

-> No.

--> REPL:

-> REPL stands for Read Eval Print Loop

-> It specifies a computer environment like a window console or Unix/Linux shell where you can enter a command, and the computer responds with an output.

-> Following are the terms used in REPL with their defined tasks:

-> Read: It reads user's input; parse the input into JavaScript data-structure and stores in memory.

-> Eval: It takes and evaluates the data structure.

-> Print: It is used to print the result.

-> Loop: It loops the above command until user press ctrl-c twice to terminate.

--> Npm:

-> Following are the two main functionalities of npm:

-> Online repositories for node.js packages/modules which are searchable on search.nodejs.org

-> Command line utility to install packages, do version management and dependency management of Node.js packages.

--> Buffer class:

-> The Node.js provides Buffer class to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

-> It is a global class and can be accessed in an application without importing a buffer module.

-> Buffer class is used because pure JavaScript is not compatible with binary data.

-> So, when dealing with TCP streams or the file system, it's necessary to handle octet streams.

--> Assert:

-> The Node.js Assert is a way to write tests(test cases).

--> Streams:

-> The Streams are the objects that facilitate you to read data from a source and write data to a destination.

-> Types:

-> Readable: This stream is used for reading operations.

-> Writable: This stream is used for write operations.

-> Duplex: This stream can be used for both reading and write operations.

-> Transform: It is a type of duplex stream where the output computes according to input.

--> Event-driven programming:

-> Means as soon as Node starts its server, it initiates its variables, declares functions and then waits for an event to occur.

-> It is one of the reasons why Node.js is pretty fast compared to other similar technologies.

Express JS

--> Notes:

-> Express.js is a web framework for Node.js. It is a fast, robust and asynchronous in nature.

-> You can assume express as a layer built on the top of the Node.js that helps manage a server and routes.

-> It provides a robust set of features to develop web and mobile applications.

--> Features:

-> It can be used to design single-page, multi-page and hybrid web applications.

-> It allows to setup middlewares to respond to HTTP Requests.

-> It defines a routing table which is used to perform different actions based on HTTP method and URL.

-> It allows to dynamically render HTML Pages based on passing arguments to templates.

--> Why use Express:

-> Ultra fast I/O

-> Asynchronous and single threaded

-> MVC like structure

-> Robust API makes routing easy

--> Body-parser:

-> This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.

--> Cookie-parser:

-> It is used to parse Cookie header and populate req.cookies with an object keyed by the cookie names.

--> Multer:

-> This is a node.js middleware for handling multipart/form-data.

--> GET Request:

-> GET and POST both are two common HTTP requests used for building REST API's.

-> GET requests are used to send only limited amount of data because data is sent into header.

-> POST requests are used to send large amount of data because data is sent in the body.

--> Cookies Management:

-> Cookies:

-> Small piece of information i.e. sent from a website and stored in user's web browser when user browses that website.

-> npm install cookie-parser

-> Import cookie-parser into your app:

```
->  var express = require('express');
    var cookieParser = require('cookie-parser');
    var app = express();
    app.use(cookieParser());
```

-> Define a route:

-> Adding Cookie:

```
app.get('/cookie',function(req, res){
    res.cookie('cookie_name' , 'cookie_value').send('Cookie is set');
});
```

-> Adding Cookie with expiry:

```
-> res.cookie(name, 'value', {expire: 360000 + Date.now()});
```

-> Deleting Existing Cookies

```
app.get('/clear_cookie_foo', function(req, res){
    res.clearCookie('foo'); // This is the main line
    res.send('cookie foo cleared');
});
```

--> File Upload:

-> In Express.js, file upload is slightly difficult because of its asynchronous nature and networking approach.

-> It can be done by using middleware to handle multipart/form data.

-> There are many middleware that can be used like multer, connect, body-parser etc.

--> Middleware:

-> In stack, middleware functions are always invoked in the order in which they are added.

-> Examples:

-> Application-level middleware

-> Router-level middleware

-> Error-handling middleware

-> Built-in middleware

-> Third-party middleware

--> Scaffolding:

-> Scaffolding is a technique that is supported by some MVC frameworks.

-> It is mainly supported by the following frameworks:

-> Ruby on Rails, Express Framework, Django, Laravel, CodeIgniter, ASP.NET etc.

-> Command: npm install express-scaffold

-> Non-programming Def:

-> Temporary structure on the outside of a building, made of wooden planks and metal poles, used by workmen while building, repairing, or cleaning the building.

--> Template Engine:

-> A template engine facilitates you to use static template files in your applications.

-> At runtime, it replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.

-> Example:

-> Pug (formerly known as jade), ejs, jazz, JUST, toffee, etc.

--> Test Cases:

-> Requirements/Installations:

-> Install Mocha, Install Chai, Prerequisites, Project Setup, The server, Write the test case, Run the test case

-> A Test case basically a line of code where we write the condition to test the another line of code.

-> Mocha:

-> Used for testing.

-> It is a javascript framework for nodejs.

-> Mocha is asynchronous.

-> Mocha is used to build an environment where we can write and run our test cases.

-> We can also use our assertion libraries under the mocha environment, here we will use "Chai".

-> Chai:

-> BDD (Behavior Driven Development) / TDD (Test Driven Development) assertion library.

-> Provide a better way to write the test cases under mocha environment.

-> Why Chai:

-> Actually Mocha provide the environment for making our test, But we need to test our API and our API using http calls like GET, PUT, DELETE, POST etc.

-> So we need a assertion library to fix this challenge. Chai helps us to determine the output of this test case.

-> Interfaces:

-> Should:

-> chai.should(); // function calling

-> foo.should.be.a('string');

-> foo.should.equal('bar');

-> foo.should.have.lengthOf(3);

-> foo.should.have.property('flavors').with.lengthOf(3)

-> Expect:

-> var expect = chai.expect; // requiring

-> expect(foo).to.be.a.('string')

-> expect(foo).to.equal('bar');

-> expect(foo).to.have.lengthOf(3);

-> expect(foo).to.have.property('flavors').with.lengthOf(3);

-> Assert:

-> var assert = chai.assert; // requiring

-> assert.typeOf(foo, 'string');

-> assert.equal(foo,'bar');

-> etc.

-> Prerequisites:

-> Installed Nodejs.

-> Installed Mocha (Run command: "npm install -g mocha")

-> All test cases will be placed within the test folder.

-> Install dependencies:

-> Chai, Chai-http, Mocha, Express

-> Writing Test Case:

-> Path: myApp — -> test — -> apitest.js

-> Example:

```
let chai = require('chai');
let chaiHttp = require('chai-http');
```

```
var should = chai.should();
chai.use(chaiHttp);
let server = require('../app');

//Our parent block
describe('Podcast', () => {
describe('/GET media', () => {
	it('it should GET all the podcast', (done) => {
	chai.request(server)
	.get('/media')
	.end((err, res) => {
			(res).should.have.status(200);
			(res.body).should.be.a('object');
			(res.body.podcasts.length).should.be.eql(1);
			done();
		});
	});
});
```

-> Explanatio of the code:

-> describe('Podcast', () => { …… }):

-> Describe function is use to describe about what we want.

It takes two parameter a string (anything that tells you about the operation) and a callback function.

->it('it should GET all the podcast', (done) => { }):

-> The "it()" used to tells us that what is going to be tested in this method.

-> Run the test case:

-> Ways:

-> Type "mocha" in the terminal.

-> Type "npm test" in the terminal.

--> Project Structure:

->

->

--> MEAN Commands:

->

Loopback 4

--> Notes:

-> 2019 ward winner in "Best API in Middleware".

-> API = secure communication between digital resuorces.

-> Remote Method = Server side handler function against each route.

-> @param.path.string('userId') = Means userId parameter is passing at the runtime.

-> DTO = Data Transfer Object

-> LoopBack 3 treated models as the 'center' of operations; in LoopBack 4, that is no longer the case.

-> StrongLoop

-> LB4 is written in TypeScript while the older versions were written in JavaScript.

-> OpenAPI Specification = Swagger Specification

->

--> Commands:

-> lb4 controller

-> lb4 datasource

-> lb4 import-lb3-models

-> lb4 model

-> lb4 relation

-> lb4 repository

-> lb4 service

-> lb4 openapi

-> lb4 discover

-> lb4 observer

-> lb4 inceptor

-> lb4 rest-crud

*-> rm src/datasources/db.datasource.**

-> lb4 uninstall-completion

-> npm i @loopback/authentication

-> pnm i @loopback/authentication-jwt

->

->

--> Naming Convention:

-> Class name: PascalCase / Title case

-> File name: kebab-case.

-> Variable name: camelCase.

--> Key Points / Guidelines:

-> Model Driven approach(Models):

-> modeling API endpoints

-> modeling domain objects

-> implementing authentication components independently

-> decorating API endpoints with authentication configurations

-> provides programming constructs for external APIs as Services

-> provides entity layer abstraction with Repository pattern

-> dependency injection to wire classes and methods with their dependencies

--> Services:

-> REST

-> SOAP

--> OpenAPI Specification (OAS):

-> REST APIs Standard

--> API implementation:

-> Controllers:

-> API Endpoints

-> These endpoints will have to authenticate incoming request, parse and validate as well as orchestrate calls to Services and Repositories.

-> Services:

-> Provide common interfaces for external APIs and services.

-> E.g: JWT Token, Password Hasaher, etc.

-> Models & Relations:

-> Represent domain objects and provide entity relationship models.

-> Repositories:

-> Represent the Entity layer for a specific model and handle all CRUD operations on the model.

-> They also use repository of other models to handle entity relations.

-> A custom method can be added to the repository class. For example, if we want to find a TodoList with a specific title from the repository level

-> LoopBack repository encapsulates your TypeScript/JavaScript Class instance and the methods that communicate with your database.

-> It is an interface to implement data persistence.

--> Dependency injection:

-> Used to wire dependencies into constructors, class properties and methods.

-> The UserController needs to connect to a user service to verify the user credentials and a JWT service to create a token.

-> Having these dependencies loosely coupled with the UserController will help developers of the Login use case with separation of duties and inject mock services for rapid testing.

--> API Security:

-> Security implementations in LoopBack can be created as separate Authentication strategies and the @authenticate decorator can be used to define the authentication strategy of a particular endpoint.

-> Example:

-> A JWT Authentication Strategy is implemented with a name property jwt.

-> whoAmI remote method in UserController is decorated with authenticate('jwt') to indicate the API endpoint is authenticated with json web tokens.

--> Application Structure Overview:

-> Controller

-> Repository

-> Model

-> In-memory datasource

-> Service Proxy

-> Geo Coding Service REST APIs

--> Files with Purpose:

-> package.json:

-> Your application's package manifest.

-> tsconfig.json

-> .eslintrc.js

-> .prettierrc

-> README.md:

-> The Markdown-based README generated for your application.

-> LICENSE:

-> A copy of the MIT license. If you do not wish to use this license, please delete this file.

-> src/application.ts:

-> The application class, which extends RestApplication by default.

-> This is the root of your application, and is where your application will be configured.

-> It also extends RepositoryMixin which defines the datasource.

-> src/index.ts:

-> The starting point of your microservice.

-> This file creates an instance of your application, runs the booter, then attempts to start the RestServer instance bound to the application.

-> src/sequence.ts:

-> An extension of the Sequence class used to define the set of actions to take during a REST request/response.

-> src/controllers/README.md:

-> Provides information about the controller directory, how to generate new controllers, and where to find more information.

-> src/__tests__/:

-> Please place your tests in this folder.

-> .mocharc.json:

-> Configuration for running your application's tests.

--> Intergration with Geo-Coding Service:

-> https://loopback.io/doc/en/lb4/todo-tutorial-geocoding-service.html

->

--> JWT:

-> Json Web Token

-> Secure APIs end points

-> Installation:

-> npm i --save @loopback/authentication @loopback/authentication-jwt

->

--> Execution Pipeline Architecture:

-> https://loopback.io/doc/en/lb4/Crafting-LoopBack-4.html

-> LoopBack uses Express routing and middleware as the plumbing to a request/response pipeline for API use cases, such as authentication, authorization, and routing.

-> API client (SDK) => API Explorer => API Specs(Swagger) => REST Endpoints => Middleware => Auth => Remoting => ACL =>

API implementation(Model & Relations) => Datasources => Backend Resources(Connector(db, rest api, soap web services, gRPC Microservices, etc))

--> Ecosystem:

-> Loopback Core:

-> API Developers

-> Extension Developers

-> Platform Developers

-> LB Maintainers & Contributers

--> Design Patterns:

-> DRY (Don't Repeat Yourself) = Reusable Microservices

-> YAGNI (You Aint't Gonna Need It) = Build minimum features and add more later if necessary

--> Why Express behind the scenes:

-> Because of the vast ecosystem of express.

-> Some of the gaps between what Express offers and LoopBack's needs are:

-> Lack of extensibility:

-> Express is only extensibile via middleware. It neither exposes a registry nor provides APIs to manage artifacts such as middleware or routers.

-> Lack of composability:

-> Express is not composable.

-> For example, app.use() is the only way to register a middleware. The order of middleware is determined by the order of app.use.

-> Lack of declarative support:

-> In Express, everything is done by JavaScript.

-> In contrast, LoopBack is designed to facilitate API creation and composition by conventions and patterns as best practices.

-> Circling back to Express with a twist:

-> The main purpose of LoopBack is to make API creation easy, interacting with databases, services, etc., not middleware for CORS, static file serving, etc.

-> Deep dive into LoopBack 4 extensibility:

-> There are several key pillars to make extensibility a reality for LoopBack 4.

-> Pillars:

-> Context, the IoC container to manage services

-> Dependency injection to facilitate composition

-> Decorators to supply metadata using annotations

-> Components as the packaging model to bundle extensions

--> Application:

-> The Application class is the central class for setting up all of your module's components, controllers, servers and bindings.

-> The Application class extends Context and provides the controls for starting and stopping itself and its associated servers.

-> NOTE: When using LoopBack 4, we strongly encourage you to create your own subclass of Application to better organize your configuration and setup.

--> Component:

->

->

--> Context:

-> An abstraction of all state and dependencies in your application.

-> LoopBack uses context to manage everything

-> A global registry for anything/everything in your app (all configs, state, dependencies, classes, etc)

-> An inversion of control container used to inject dependencies into your code

-> Bindings:

-> controller.UserController

-> repositories.UserRepository

-> services.PasswordHasher

-> datasources.MySQL

-> You can use the context as a way to give loopback more "info" so that other dependencies in your app may retrieve it. It works as a centralized place/ global built-in/in-memory storage mechanism.

-> Service UserController => (binds) => controller.UserController

-> repository.UserRepository => (injects/get) => Service UserController

-> Levels:

-> Application-level context (Global):

-> Stores all the initial and modified app states throughout the entire life of the app (while the process is alive)

-> Generally configured when the application is created (though the context may be modified while running)

-> Server-level context:

-> A child of application-level context.

-> Holds configuration specific to a particular server instance.

-> our application will typically contain one or more server instances, each of which will have the application-level context as its parent. This means that any bindings that are defined on the application will also be available to the server(s),

unless you replace these bindings on the server instance(s) directly.

-> Request-level context (Request):

->

->

-> Heirarchy:

-> Application (Root Context) => (parent of) => RestServer (Owner Context) => (parent of) => RequestContext (Resolution Context)

-> Example:

-> Application = controller.pingController

-> RestServer = RestBinding.SEQUENCE, SequenceActions.FIND_ROUTE

-> RequestContext = RestBinding.Http.CONTEXT, CoreBinding.CONTROLLER_CURRENT

-> Events:

-> bind

-> unbind

-> error

-> Context observers:

->

->

-> Context View:

->

->

--> Binding:

-> Represents items within a Context instance.

-> A binding connects its value to a unique key as the address to access the entry in a context.

-> Attributes:

-> key: Each binding has a key to uniquely identify itself within the context

-> scope: The scope controls how the binding value is created and cached within the context

-> tags: Tags are names or name/value pairs to describe or annotate a binding

-> value: Each binding must be configured with a type of value provider so that it can be resolved to a constant or calculated value

--> Dependency Injection:

-> Technique where the construction of dependencies of a class or function is separated from its behavior, in order to keep the code loosely coupled.

-> Example:

-> The Sequence Action authenticate in @loopback/authentication supports different authentication strategies (e.g. HTTP Basic Auth, OAuth2, etc.).

->

->

--> Interceptor:

-> Reusable functions to provide aspect-oriented logic around method invocations.

-> Use Cases Examples:

-> *Add extra logic before / after method invocation, for example, logging or measuring method invocations.*

-> *Validate/transform arguments*

-> *Validate/transform return values*

-> *Catch/transform errors, for example, normalize error objects*

-> *Override the method invocation, for example, return from cache*

->

->

->

--> *Life Cycle Events and Onservers:*

-> *boot():*

->*!booted -> booting -> booted*

-> *start():*

-> *!started -> starting -> started*

-> *stop():*

-> *started -> stopping -> stopped*

--> *Services:*

-> *Refer to an object with methods to perform local or remote operations.*

-> *Such objects are resolved from bindings within a LoopBack 4 context.*

-> *We support three types of services:*

-> *proxy: A service proxy for remote REST/SOAP/gRPC APIs*

-> *class: A TypeScript class that can be bound to the application context*

-> *provider: A TypeScript class that implements Provider interface and can be bound to the application context*

-> *Generate services using CLI:*

-> *lb4 service (generate code for local and remote services)*

-> *Register services by code (instead of command):*

-> *Bind a service class or provider = const binding = app.service(MyService);*

-> *Bind a service class or provider with an interface identified by a string = const binding = app.service(MyService, {interface: 'MyService'});*

-> *Bind a service with an interface identified by a symbol = ...code here...*

-> *Inject service instances:*

->

->

--> *Booter:*

-> *What does Booting an Application mean:*

-> *Discovering artifacts automatically based on a convention (a specific folder containing files with a given suffix)*

-> *Processing those artifacts (this usually means automatically binding them to the Application's Context)*

-> *Articraft:*

-> *An artifact is any LoopBack construct usually defined in code as a Class.*

-> *LoopBack constructs include Controllers, Repositories, Models, etc.*

-> *Usage:*

-> *@loopback/cli:*

-> *New projects generated using @loopback/cli or lb4 are automatically enabled to use @loopback/boot for booting the Application using the conventions followed by the CLI.*

-> *Adding to existing project:*

-> *Use "BootMixin" to add Boot to your Project manually.*

--> *Decorators:*

-> *A decorator allows you to annotate or modify your class declarations and members with metadata.*

-> *Decorators give LoopBack the flexibility to modify your plain TypeScript classes and properties in a way that allows the framework to* *better understand how to make use of them, without the need to inherit base classes or add functions that tie into an API.*

-> Built-in decorators in lb4:

-> OpenAPI Decorators:

-> Route Decorators:

-> Used to expose controller methods as REST API operations.

-> Types:

-> API Decorator:

-> Syntax: @api(spec: ControllerSpec)

-> @api is used when you have multiple Paths Objects that contain all path definitions of your controller.

-> NOTE: api specs defined with @api will override other api specs defined inside the controller.

-> Operation Decorator:

-> Syntax: @operation(verb: string, path: string, spec?: OperationObject)

-> It exposes a Controller method as a REST API operation and is represented in the OpenAPI spec as an Operation Object.

-> Commonly-used Operation Decorators:

-> Syntax: @get(path: string, spec?: OperationObject)

-> Same Syntax for decorators @post , @put , @patch , @del

-> You can call these sugar operation decorators as a shortcut of @operation.

-> For example (below both are same):

-> @get('/greet', spec)

-> @operation('GET', '/greet', spec)

-> Parameter Decorator:

-> @param is applied to controller method parameters to generate an OpenAPI parameter specification for them.

->

->

-> Dependency Injection Decorator:

-> @inject

-> @inject.getter

-> @inject.setter

-> @inject.binding

-> @inject.tag

-> @inject.view

-> @inject.context

-> Authentication Decorator:

-> Syntax:

-> single strategy: @authenticate(strategyName)

-> multiple strategies: @authenticate(strategyName1, strategyName2)

-> Method Level Decorator:

-> Example:

-> @authenticate('BasicStrategy')
@get('/whoami') // means above method

-> Class Level Decorator:

-> @authenticate('BasicStrategy')
export class WhoAmIController {} // means above class

-> Multiple Strategies:

->

->

-> Authorization Decorator:

-> Syntax: @authorize({resource: 'order', scopes: ['create']})

-> The authorization decorator is used to provide access control metadata.

-> Part of the component @loopback/authorization.

-> It is applied to controller members and is used to specify who can perform which operations to the protected resource.

-> The @authorize decorator takes in an object in type AuthorizationMetadata.

-> A full list of the available configuration properties are:

-> allowedRoles/deniedRoles: Define the ACL based roles. It should be an array of strings.

-> voters: Supply a list of functions to vote on a decision about a subject's accessibility. A voter is a method or class level authorizer.

-> resource: Type of the protected resource, such as customer or order.

-> scopes: An array of the operations against the protected resource, such as get or delete.

-> skip: A boolean value to mark an endpoint/a controller skips the authorization.

-> Method Level Decorator:

```
-> @authorize({
            allowedRoles: ['everyone'],
            scopes: ['create'],
            resource: 'order',
      })
      async placeOrder(order: Order) {}
```

-> Class Level Decorator:

```
-> @authorize({allowedRoles: ['ADMIN']})
   export class MyController {}
```

-> Shortcuts:

-> allow, allowAll, alloAllExcept, allowAuthenticated, deny, denyAll, denyAllExcept, denyUnauthenticated, scope, skip, vote.

-> Service Decorator:

-> Syntax: @service(serviceInterface: ServiceInterface, metadata?: InjectionMetadata)

-> @service injects an instance of a given service from a binding matching the service interface.

-> The service interface can be a class.

-> Example:

```
class MyController {
      constructor(@service(MyService) public myService: MyService) {}
}
```

-> If the service is modeled as a TypeScript interface, we need to use a string or symbol to represent the interface as TypeScript interfaces cannot be reflected at runtime.

-> Example:

```
const MyServiceInterface = 'MyService';
class MyController {
      constructor(@service(MyServiceInterface) public myService: MyService) {}
}
```

-> Repository Decorators:

-> Repository decorators are used for defining models (domain objects) for use with your chosen datasources and for the navigation strategies among models.

-> Model Decorators:

-> Model is a class that LoopBack builds for you to organize the data that shares the same configurations and properties.

-> You can use model decorators to define a model and its properties.

-> Syntax: @model(definition?: ModelDefinitionSyntax)

-> Property Decorator:

-> Syntax: @property(definition?: PropertyDefinition)

-> The property decorator defines metadata for a property on a Model definition.

-> Repository Decorator:

->

->

-> Relation Decorators:

-> The relation decorator defines the nature of a relationship between two models.

-> Syntax: @relation

-> Types:

-> BelongsTo Decorator:

-> Syntax: @belongsTo(targetResolver: EntityResolver<T>, definition?: Partial<BelongsToDefinition>)

-> HasOne Decorator:

-> Syntax: @hasOne(targetResolver: EntityResolver<T>, definition?: Partial<HasOneDefinition>)

-> HasMany Decorator:

-> Syntax: @hasMany(targetResolver: EntityResolver<T>, definition?: Partial<HasManyDefinition>)

-> Other Decorators:

-> The following decorators are not implemented yet.

-> @embedsOne, @embedsMany, @referencesOne, @referencesMany

-> Mixin:

-> It is a commonly used JavaScript/TypeScript strategy to extend a class with new properties and methods.

-> A good approach to apply mixins is defining them as sub-class factories. Then declare the new mixed class as: class MixedClass extends MixinFoo(MixinBar(BaseClass)) {}

-> Examples:

-> TimeStamp Mixin, Logger Mixin

-> NOTE: A TypeScript limitation prevents mixins from overriding existing members of a base class. Hence the need for // @ts-ignore

--> Rest APIs:

-> Server:

-> They typically listen for requests on a specific port, handle them, and return appropriate responses.

-> Servers in LoopBack 4 are used to represent implementations for inbound transports and/or protocols such as REST over http, gRPC over http2, graphQL over https, etc.

-> A single application can have multiple server instances listening on different ports and working with different protocols.

-> Common tasks:

-> Enable HTTPS

-> Customize how OpenAPI spec is served

-> Usage:

-> LoopBack 4 offers the @loopback/rest package out of the box, which provides an HTTP/HTTPS-based server called RestServer for handling REST requests.

->In order to use it in your application, your application class needs to extend RestApplication to provide an instance of RestServer listening on port 3000.

-> Sequence:

-> A Sequence is a series of steps to control how a specific type of Server responds to incoming requests.

-> Each types of servers, such as RestServer, GraphQLServer, GRPCServer, and WebSocketServer, will have its own flavor of sequence.

-> The sequence represents the pipeline for inbound connections.

-> Each server type has a default sequence.

-> *It's also possible to create your own Sequence to have full control over how your Server instances handle requests and responses.*

-> *The sequence itself is basically a named middleware chain.*

-> *Each middleware serves as an action within the sequence.*

-> *The handle function executes registered middleware in cascading fashion.*

-> *Types:*

-> *Middleware based*

-> *Action based:*

-> *NOTE: Action-based sequence is now being phased out.*

-> *Routing request:*

->

->

-> *Parsing request:*

->

->

-> *Middleware:*

-> *Node.js web frameworks such as Express and Koa use middleware as the basic building blocks to compose a pipeline of functions that handles HTTP requests and responses.*

-> *Why not use Express middleware as-is:*

-> *You may wonder why we don't expose Express middleware directly.*

-> *There are some gaps in Express middleware that we would like to close to better align with LoopBack's architecture.*

-> *Express middleware are one-way handlers that mostly process requests and there is no first-class facility to handle responses.*

We would love to support Koa style middleware that use async/await to allow cascading behaviors.

-> *Express middleware are added by the order of app.use() and it's hard to contribute middleware from other modules.*

-> *Express does not allow dependency injection. We would like to enable injection of middleware configurations.*

-> *Controller:*

-> *A Controller is a class that implements operations defined by an application's API.*

-> *It implements an application's business logic and acts as a bridge between the HTTP/REST API and domain/database models.*

-> *Request/Response cycle:*

-> *Loopback 3:*

-> *Request => initial => session => auth => parse => routes:before => Express:middleware => components => Boot script*

=> routes => routes:after => files => final => Response

-> *Loopback 4:*

-> *Request => FindRoute => ParseParams => InvokeMethod => Controller Methods => Repository => Datasource => Database => send/reject => Response*

=> Non-Controller Methods => send/reject => Response

--> *Data Access:*

-> *Model:*

->

->

-> *Relations:*

-> *HasMany*

-> *BelongsTo*

-> *HasOne*

-> *HasMany Through*

-> *Repository:*

->

->

-> DataSource:

->

->

--> Best Practices:

-> Defining the API using code-first approach

-> Defining your testing strategy

-> Testing your application

--> Services:

-> proxy: A service proxy for remote REST/SOAP/gRPC APIs

-> class: A TypeScript class that can be bound to the application context

-> provider: A TypeScript class that implements Provider interface and can be bound to the application context

--> Lodash:

-> omit

->

->

->

Coding Standards & Guidelines

--> General Points:

-> Declare array in a single line.

-> var a = ['hello', 'world'];

-> Decalre objects in multiline.

-> var b = {

good: 'code',

'is generally': 'pretty',

};

-> Use the === operator.

-> Check not condition first in if check and this should be a base case or termination case.

-> Leave blank space between two different methods or functions.

-> Use 'Requires' at top.

-> Use helper libraries.

->

->

--> Formatting:

-> 4 spaces for indentation:

-> Use 4 spaces for indenting your code and swear an oath to never mix tabs and spaces — a special kind of hell awaits otherwise.

-> Newlines:

-> Use UNIX-style newlines (\n), and a newline character as the last character of a file. Windows-style newlines (\r\n) are forbidden inside any repository.

-> No trailing white space:

-> You clean up any trailing white space in your .js files before committing.

-> Use semicolons.

-> 80 characters per line.

-> Use single quotes.

-> Opening braces go on the same line.

-> Declare one variable per var statement.

--> Naming Conventions:

-> Use lowerCamelCase for variables, properties and function names.

-> Use UpperCamelCase for class names.

-> Use UPPERCASE for Constants.

-> File names must be all lowercase and may include underscores (_) or dashes (-)

--> Functions:

-> Write small functions.

-> Return early from functions:

-> To avoid deep nesting of if-statements, always return a function's value as early as possible.

-> Right Example:

```
-> function isPercentage(val) {
        if (val < 0) {
            return false;
        }
        if (val > 100) {
            return false;
        }
        return true;
        }
```

-> Wrong Example:

```
-> function isPercentage(val) {
    if (val >= 0) {
        if (val < 100) {
        return true;
        } else {
        return false;
        }
    } else {
        return false;
    }
}
```

--> High CPU consuption program:

-> If your program is using high cpu consumption like using multiple db calls, for this node hs is not suitable

because it is a single threaded (single CPU core). Use other languages like ROR or Django for this.

--> Constants:

-> increase speed (specially in compile time languages).

--> Camel Cases:

-> Upper Camel Case:

-> let MyVar = 0;

-> we can also say it title case.

-> Lower Camel Case:

-> let myVar = 0;

--> Error Handling Practices:

-> Use Async-Await or promises for async error handling.

-> Use only the built-in Error object.

-> Wrong: throw ("Name is required");

-> Correct: throw new Error("Name is required");

--> Use a mature logger to increase error visibility:

-> Winston, Bunyan, Log4js or Pino, will speed-up error discovery and understanding. So forget about console.log.

-> Arqam has used "loopback-component-logger" logger.

--> APM:

-> Action Per Minute.

-> Used for performance analysis of the App.

--> ESLint:

-> Linting:

->Linting is the process of running a program that will analyse code for potential errors.

-> A program that would go through your C code and identify problems before you compiled.

-> Process of checking the source code for Programmatic as well as Stylistic errors.

-> Interpreted languages like Python and JavaScript benefit greatly from linting, as these languages don't have a compiling phase to display errors before execution.

--> OOP in javascript:

-> The two important principles with OOP in JavaScript are Object Creation patterns (Encapsulation) and Code Reuse patterns (Inheritance)

-> Use the Combination Constructor/Prototype Pattern to implement encapsulation.

-> Patterns: Constructor, Prototype, Combination Constructor/Prototype Pattern, etc.

--> Object Creation:

-> Ubiquitous Object Literal:

```
-> var myObj = {
        name: "Nikki",
        city: "New Delhi",
        loves: "so many"
    }
```

-> Prototype Pattern:

```
    -> function myFun() {};
        myFun.prototype.name = "Nikki";
        myFun.prototype.city = "New Delhi";
        var myFun1 = new myFun();
        console.log(myFun1.name); //Nikki
```

-> Constructor Pattern:

```
    -> function myFun (name, city, loves){
            this.name = name;
            this.city = city;
            this.loves= loves;
        }
        var myFun1 = new myFun ("Nikki", "New Delhi", "so much things");
        console.log(myFun1.name); //Nikkia
```

-> Combination Constructor/Prototype Pattern:

```
    -> var prop1 = "one"; //private
        this.prop2= "two" //public
        var tc = new test();
        var tp1 =tc.prop1; //undefined: because prop1 is private;
```

--> Classes in Javascript:

-> Remote methods / functions are also classes in Js.

-> It is a bad idea to use classes in JavaScript, and what are some of the alternatives.

-> OOP is fading away in general.

-> JavaScript is not an object-oriented language, it was not designed to be one, the notion of classes is absolutely not applicable to it.

-> JavaScript has no idea what classes are.

-> While everything in JS is indeed an object (even the functions are objects), these objects are different from the ones in Java or C#.

-> There wes classes concept in js before EC6.

-> Why using classes in JS is a bad idea:

-> Binding issues:

-> Performance issues:

-> Private variables:

-> One of the great advantages and the main reasons for classes in the first place, private variables, is just non-existent in JS.

-> Strict hierarchies:

-> Classes introduce a straight top-to-bottom order and make changes harder to implement, which is unacceptable in most JS applications.

-> Because the React team tells you not to:

-> While they did not explicitly deprecate the class-based components yet, they are likely to in the near future.

--> Design Patterns:

-> A design pattern is a general, reusable solution to a commonly occurring problem.

-> Speed up the code.

-> Testing becomes easy.

-> Types:

-> Immediately Invoked Function Expressions (IIFE):

-> It allows you to define and call a function at the same time.

-> Helps in complex code.

-> Increases the speed because, as the name suggests, you're executing the function as soon as you define it.

-> Factory method pattern:

-> Very popular.

-> It acts as a tool you can implement to clean your code up a bit.

-> Doesn't require us to use a constructor but provides a generic interface for creating objects.

-> This pattern can be really useful when the creation process is complex.

-> Singleton pattern:

-> Oldie but a goodie.

-> It helps you keep track of how many instances of a class you're instantiating.

-> Restrict the number of instantiations of a "class" to one.

-> Example:

-> We 'require' something. It does not matter how many times you will require this module in your application; it will only exist as a single instance.

-> Observers:

-> ReactiveX - bringing the developers a new powerful tool, the Observable.

-> Observables produces multiple values called a stream (unlike promises that return one value).

-> Dependency Injection:

-> A software design pattern in which one or more dependencies (or services) are injected, or passed by reference, into a dependent object.

-> Streams:

-> They are better at processing bigger amounts of flowing data, even if they are bytes, not objects.

--> Errors and Error Codes:

-> Types:

-> Standard JavaScript errors:

-> Examples: RangeError , ReferenceError , and TypeError.

-> System-level errors from the underlying operating system:

-> These might occur due to invalid IO calls or insufficient memory issues.

-> User errors generated by application code:

->

-> Assertion Errors:

-> Special error type triggered whenever Node.js detects a logic violation.

-> Error Codes:

-> You can view list of error codes by tyoing following commands in sequence on the terminal:

-> node

-> http.STATUS_CODES

-> Series:

-> 100-102

-> 200-207

-> 300-307

-> 400-431

-> 500-511

-> Top Codes:

-> 200: OK

-> 202: Accepted

-> 204: No Content

-> 400: Bad Request

-> 401: Unauthorized

-> 403: Forbidden

-> 404: Not Found

-> 405: Method Not Allowed

-> 500: Internal Server Error

-> 502: Bad Gateway

-> 504: Gateway Time-out

-> 511: Network Authentication Required

Node.js v15.10.0 Documentation

--> Async hooks:

-> Async Hooks are a core module in Node. js that provides an API to track the lifetime of asynchronous resources in a Node application.

-> An asynchronous resource can be thought of as an object that has an associated callback.

-> Example:

-> Promises, Timeouts, TCPWrap, UDP etc.

--> open API (public API):

-> an application programming interface made publicly available to software developers.

-> types: rest and soap

-> Security:

-> as it is an open source, there is a need to secure it.

-> Transport Layer Security (TLS) can encrypt data sent across a network.

-> For authenticating the identity of the caller, SSL certificates prove useful, and any number of back-end authentication mechanisms can map users to the private data that is associated with a given user principle.

-> Open vs. closed APIs:

->

->

--> Buffer:

-> Class in Node. js is designed to handle raw binary data.

-> Each buffer corresponds to some raw memory allocated outside V8.

-> Buffers act somewhat like arrays of integers, but aren't resizable and have a whole bunch of methods specifically for binary data.

-> Pure JS doesn't handles straight binary data very well.

-> Example:

-> var buf = Buffer.from('abc');

console.log(buf); // <buffer 61 62 63>

--> C++ addons:

-> Addons are dynamically-linked shared objects written in C++.

-> *Addons provide an interface between JavaScript and C/C++ libraries.*

->

--> *Cluster:*

-> *Like we have 4 processors (quad core).*

-> *Boost Node App Performance & Stability.*

-> *Using round robin algos*

-> *How to handle tokens in clustering?*

-> *There will be a primary cluster*

--> *Concurrent VS Parallel:*

-> *concurrent: multiple things heppening in the same period of time.*

-> *parallel: multiple things are happening at the same moment.*

Node JS Advanced Concepts

--> *JS Code:*

-> *Node JS (stores the dependencies of javascript)*

-> *V8 (executes js code) (http + crypto)*

-> *libuv (gives access to networking and file system, etc) (fs + path)*

--> *JS Code => Node JS => process.binding() => V8 => Node js C++ side => libuv*

--> *Process:*

-> *Instance of the program that is being executing.*

-> *A single process can have multiple threads within it.*

-> *Each program has only one event loop.*

--> *Improving Node JS performance:*

-> *Use Node in 'Cluster Mode' (Recommended)*

-> *Use Worker Thread (Experimental)*

--> *Cluster Management:*

-> *Single Thread (having nodie js server)*

-> *Single Thread (having nodie js server)*

-> *Single Thread (having nodie js server), etc.*

--> *Benchmark:*

-> *Means evaluating (something) by comparison with a standard.*

--> *Web worker:*

-> *When executing scripts in an HTML page, the page becomes unresponsive until the script is finished.*

-> A web worker is a JavaScript that runs in the background, independently of other scripts, without affecting the performance of the page.

You can continue to do whatever you want: clicking, selecting things, etc., while the web worker runs in the background.

--> Testing:

-> Unit Testing:

-> Assert that one piece of your app is working the way you expect.

-> Integration Testing:

-> Make sure that multiple 'units' work together correctly.

--> Chromium:

-> Subset of chrome and edge.

-> also used in puppeteer

--> Captcha:

-> When we run multiple scripts, google starts observing us and shows the captha.

--> Middlewares:

-> Morgan:

-> HTTP request logger middleware for Node.js that generates logs for each API request.

-> The best part is that you can either use a predefined format or create a new one based on your needs.

-> Winston:

-> logger library

-> Helmet:

-> Helmet is a security middleware that protects Express.js apps by setting various HTTP headers.

->

--> Cors:

-> CORS stands for cross-origin resource sharing. It is used to enable and configure CORS in Express.js apps.

-> Imagine you have a full-stack app with a React frontend running on port 3000 and an Express backend server running on port 8000.

A request comes from the client (i.e., the React frontend) to the backend Express server, but your request will most likely fail since it

is coming from a different origin than the Express server.

--> ES6:

-> Arrows:

-> Unlike functions, arrows share the same lexical this as their surrounding code.

-> Do not have this

-> Do not have arguments

-> Can't be called with new

-> They also don't have super, but we didn't study it yet.

-> Rest VS Spread:

-> Rest parameter: collects all remaining elements into an array. (like parameters of a function)

-> Spread operator: allows iterables(arrays / objects / strings) to be expanded into single arguments/elements.

-> like arguments of a function

-> like newArr = ["joykare", ...arr];

-> argument vs parameter:

-> argument = value passing to the method

-> parameter = variables in method signature

-> Arguments keyword:

-> Before rest parameters existed, to get all the arguments in a function we used arguments which is an array-likeobject.

-> Example:

```
-> function someFunction() {
        return arguments; // this will return the arguments sends while catting this function
    }
```

-> Copying arrays:

-> arr1 = [...arr]

-> Map fucntion:

-> array.map(function(currentValue, index, arr), thisValue)

-> Primitive VS non-primitive:

-> Primitive = immutable = not changeable = other than object type = int, bigint, string, etc

-> Example:

-> b = 2; a = b; b++; // value of a will be 2 not 3. means old value of b not the lates value.

-> Sets:

-> unique elements.

-> pass array/iterable in paramter of Set function.

-> Example:

-> new Set([2,2,3]);// returns array [2,3]

-> Methods:

-> add, has, size, forEach, delete, clear

-> Browser:

-> Location:

-> contains info/objects like host, hostname, pathnanme, href, reload, origin, etc.

-> Template/String Literals:

-> added in backtick like ${expression}

-> Classes:

->

-> Try-Catch-Finally:

-> Where to use it:

-> one really needs to use a try-catch is around JSON. parse()

-> All of the other methods use either the standard Error object through the first parameter of the callback or emit an error event.

-> Best practice is actually to not use try-catch blocks at all.

-> the best practice should be don't throw exceptions for any kind of error, unless it's a blocking error.

-> Promise:

-> A promise is just an enhancement to callback functions in Node.js.

-> It is to avoid call back hell.

-> The promise is kept and otherwise, the promise is broken.

-> Fetch VS Axios:

-> You might need to fetch data from or post data to an external web server or API.

-> Axios are the stand-alone third party package that can be easily installed where fetch usually installed in the modern browsers and hence no istallation is required.

-> Axios performs automatic transforms of JSON data where Fetch is a two-step process when handling JSON data - first, to make the actual request; second, to call the .json() method on the response.

-> Axios has wide browser support.

-> Stringify VS toString:

-> These are close but not quite the same.

-> json.stringify text can be converted back to the json using JSON.parse

-> Example:

-> When obj is an array like [1,2,3] then

-> .toString() gives: "1,2,3"

-> JSON.stringify gives: "[1,2,3]"

-> Example:

-> When obj is an object like obj = { a: 'a', '1': 1 } then

-> obj.toString() gives: "[object Object]"

-> JSON.stringify(obj) gives: "{"1":1,"a":"a"}"

-> Transpilers/source-to-source compilers:

-> Tools that read source code written in one programming language, and produce the equivalent code in another language.

-> Languages you write that transpile to JavaScript are often called compile-to-JS languages, and are said to target JavaScript.

-> Babel:

-> Babel is a JavaScript transpiler that converts edge JavaScript into plain old ES5 JavaScript that can run in any browser (even the old ones)

-> Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments.

-> Convert jsx to js for compilation for the browser.

-> Webpack:

-> Webpack is a static module bundler for JavaScript applications.

-> It takes modules, whether that's a custom file that we created or something that was installed through NPM, and converts these modules to static assets.

-> Convert jsx to js for compilation for the browser.

-> Fat arrow:

-> Arrow function of ES6.

-> CommonJS (ES5):

-> Module formatting system.

-> It is a standard for structuring and organizing JavaScript code.

-> Client-side JavaScript that runs in the browser uses another standard, called ES Modules.

-> Server-side JavaScript uses CommonJS as standard.

-> Modules are very cool, because they let you encapsulate all sorts of functionality, and expose this functionality to other JavaScript files, as libraries.

-> Example:

-> const package = require('module-name')

-> In CommonJS, modules are loaded synchronously, and processed in the order the JavaScript runtime finds them.

-> This system was born with server-side JavaScript in mind, and is not suitable for the client-side (this is why ES Modules were introduced).

-> Dependency Graph:

-> Any time one file depends on another, webpack treats this as a dependency.

-> This allows webpack to take non-code assets, such as images or web fonts, and also provide them as dependencies for your application.

-> Async VS Sync:

-> Otherwise what usually happens for example in PHP or Python code is that the thread blocks until the sync operation (reading from the network, writing a file..) ends.

->

-> Module (ES6):

-> A module is just a bit of code encapsulated in a file, and exported to another file.

-> Modules focus on a single part of functionality and remain loosely coupled with other filed in an application.

-> This is because there are no global or shared variables between modules, as they only communicate via the module.exports object.

-> Any code that you want to be accessible in another file can be a module.

-> MomentJS:

-> Use to deal with date and time or time zones.

-> Urlencoded data using Axios:

-> Step1: npm install qs

-> Step2: const qs = require('qs') , const axios = require('axios') OR import qs from 'qs' , import axios from 'axios'

-> Step3:

```
axios({
        method: 'post',
        url: 'https://my-api.com',
        data: qs.stringify({
                item1: 'value1',
                item2: 'value2'
        }),
        headers: {
                'content-type': 'application/x-www-form-urlencoded;charset=utf-8'
        }
        })
```

-> Flattern the array:

-> Lodash functions:

-> The cool thing about Lodash is that you don't need to import the whole library.

-> You can use those functions individually using those packages:

-> lodash.flatten:

-> Example:

-> const flatten = require('lodash.flatten')

const animals = ['Dog', ['Sheep', 'Wolf']]

flatten(animals) //['Dog', 'Sheep', 'Wolf']

-> lodash.flattendeep

-> lodash.flattendepth

-> Native JS functions:

-> lat():

-> Example:

-> ['Dog', ['Sheep', 'Wolf']].flat() //['Dog', 'Sheep', 'Wolf']

-> ['Dog', ['Sheep', ['Wolf']]].flat() //['Dog', 'Sheep', ['Wolf']]

-> ['Dog', ['Sheep', ['Wolf']]].flat(2) //['Dog', 'Sheep', 'Wolf']

-> ['Dog', ['Sheep', ['Wolf']]].flat(Infinity) //['Dog', 'Sheep', 'Wolf']

-> flatMap():

->

-> Gulp:

-> Gulp is a command-line task runner for Node. js.

-> Gulp let us automate processes and run repetitive tasks with ease.

-> What makes Gulp different from other task runners is that it uses Node streams; piping output from one task as an input to the next.

-> Gulp is a cross-platform, streaming task runner that lets developers automate many development tasks.

-> At a high level, gulp reads files as streams and pipes the streams to different tasks. These tasks are code-based and use plugins.

The tasks modify the files, building source files into production files.

--> Style Standards:

-> Google JavaScript Style Guide

-> Airbnb JavaScript Style Guide

-> Idiomatic.JS

-> StandardJS

--> Automated Linters:

-> JSLint

-> JSHint

-> ESLint

--> How to make our modern code work on older engines that don't understand recent features yet?

There are two tools for that:

-> Transpilers.

-> Polyfills.

--> Compact table for ES6:

-> https://kangax.github.io/compat-table/es6/

--> Array VS List VS ArrayList VS LinkedList:

-> Linkedlist uses too many pointers. this is why we avoid linkedlist for optimization.

Reference Links

- *https://stackify.com/learn-nodejs-tutorials/ (ALL-NODE JS-WEBSITES-LINKS)*
- *https://javascript.info/*
- *https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Howto*
- *https://www.programiz.com/cpp-programming/public-protected-private-inheritance*
- *https://codeburst.io/a-simple-guide-to-es6-iterators-in-javascript-with-examples-189d052c3d8e*
- *https://www.toptal.com/nodejs/interview-questions*
- *https://www.aivosto.com/articles/loopopt.html#loops*
- *https://developer.mozilla.org/en-US/docs/Web/API/console // console properties*
- *https://www.perfomatix.com/nodejs-coding-standards-and-best-practices/ (coding standards)*
- *https://developer.ibm.com/technologies/node-js/ (IBM Blog)*
- *https://medium.com/javascript-in-plain-english/please-stop-using-classes-in-javascript-and-become-a-better-developer-a185c9fbede1 (classes in JS)*
- *https://m.dotdev.co/how-to-use-classes-in-node-js-with-no-pre-compilers-and-why-you-should-ad9ffd63817d*
- *https://blog.logrocket.com/design-patterns-in-node-js/ (Design Patterns)*
- *https://blog.risingstack.com/fundamental-node-js-design-patterns/ (Design Patterns)*
- *https://www.freecodecamp.org/news/rxjs-and-node-8f4e0acebc7c/ (RXJS)*
- *https://medium.com/javascript-in-plain-english/how-to-read-an-excel-file-in-node-js-6e669e9a3ce1 (reading excel file)*
- *https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/#:~:text=A%20hash%20table%20is%20a,function%2C%20hashing%20can%20work%20well. (Hash Tables)*
- *https://medium.com/preezma/node-js-event-loop-architecture-go-deeper-node-core-c96b4cec7aa4#:~:text=The%20event%20loop%20is%20what,One%20iteration%20of%20Node. (Event Loop)*
- *https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff (Event Loop)*
- *https://dev.to/sandy8111112004/javascript-var-let-const-41he (Lexical scoping VS Function scope VS Global scope)*
- *https://dev.to/sandy8111112004/javascript-introduction-to-scope-function-scope-block-scope-d11#:~:text=is%20not%20defined-,Block%20Scope,only%20within%20the%20corresponding%20block ((Lexical scoping VS Function scope VS Global scope))*
- *https://dmitripavlutin.com/javascript-scope/#2-block-scope*
- *https://blog.usejournal.com/sessionless-authentication-withe-jwts-with-node-express-passport-js-69b059e4b22c (Authentication JWT etc)*
- *https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html (Security)*
- *https://medium.com/@svsh227/write-your-first-test-case-in-your-node-app-using-mocha-5250e614feb3 (Test Cases)*
- *https://www.w3schools.com/nodejs/ref_assert.asp#:~:text=Definition%20and%20Usage,be%20used%20internally%20by%20Node. (Assert for Test Cases)*
- *https://loopback.io/doc/en/lb4/index.html (Loopback 4)*
- *http://spec.openapis.org/oas/v3.0.3 (OpenAPIs)*
- *https://nodejs.org/api/events.html*
- *https://flaviocopes.com/async-vs-sync/*