

# Basics of Node Js

**Const** --> used for passwords

**let** --> used to initialize a variable for a specific scope

**var** --> used to initialize a variable for a global scope

scope of a variable: it is the particular region of the code where function can be accessed.

## If Else Statements

fucntion example(){

    var x = 10;

    if (1){

        var y = 20;

        console.log(x); // print statement

    }console.log(y);

}

example();

command: node x.js

Output: Doesn't return error

why??

Because **var** has global scope.

Hence doesn't give errors.

function example(){

    var x = 10;

    if (1){

        let y = 20;

        console.log(x);

    }console.log(y);

}

example();

Output: Error

Because **let** has scope within a function.

Hence gives error.

What is difference between iteration and

loop?

**iteration**: no. of loops i.e. process of execution.

eg. the variable that is in the code which is running multiple times.

**loop**: code that executes multiple times.

eg. watching loop from outside

fucntion example(){

    var x = 10;

    if (1){

        let y = 20;

        console.log(x);

    }console.log(y);

    if (1) {

        //

    }

    console.log(y);

}

example();

Javascript was a mistake. It was initially created to blink text in Nightscape browser.

Use let.

```
let x = 5;  
console.log(!x);
```

Output: false

```
let x = [];  
console.log (!!x);
```

truth false converter !! used for typecasting truth and false

```
console.log(x) --> []
```

```
console.log (!!x) --> Output: true  
why?? because x is defined so true
```

```
let x = {};  
consloe.log (!!x);
```

console.log (!!x) --> Output: true  
why?? because x is defined so true

```
let x = " ";  
consloe.log (!!x);
```

console.log (!!x) --> Output: false  
why?? because strings, arrays do not  
have0 a memory allocation i.e., x is not  
defined.

```
let x = null;  
consloe.log (!!x);
```

console.log (!!x) --> Output: false  
why?? because null do not have a memory  
allocation i.e., x is not defined.

```
let x = undefined;  
console.log (!!x);
```

console.log (!!x) --> Output: false

why?? because undefined do not have a memory allocation i.e., x is not defined.

```
let x = []  
if (!!x) {  
    console.log("Yes");  
}
```

Output: Yes

Because x is true , hence if statement is executed.

```
let x = [];  
let state = !!x;
```

```
if (state) {  
    console.log("Yes");  
}
```

Debugging:

before running 1st line

x: undefined

state: undefined

after running 2nd line

x: (0) []

state: undefined

after running 3rd line

x: (0) []

state: true

after running last line

x: uncaught reference

state: uncaught reference

freed the memory at the end of the code

# Ternary Operator

shorter way of using if else statement.

```
let age = 5;  
let eligible = false  
if (age>18) {  
    eligible = true;  
}  
else {  
    eligible = false;  
}
```

shorter version:

```
eligible = age > 19 ? true : false;
```

=> this means if age > 18 than eligible = true and if age < 18 than eligible = false

# Object

: key value pair

let // safer as it is for a particular scope

```
let User = {
```

```
    name: 'John',
```

```
    age: 30
```

```
    phone: '123-456-7890',
```

```
    address: {
```

```
        street: '123 Main St',
```

```
        city: 'New York',
```

```
        state: 'NY',
```

```
        zip: '10001'
```

```
    },
```

```
    hobbies: ['movies', 'music'],
```

```
    magic_number: function (x) {
```

```
        return x * x;
```

```
    }
```

```
};
```

```
console.log(User.name);
```

Output: John

```
console . log ( User [ 'name' ]); // not valid  
json
```

```
console . log ( User . address . state );
```

Output: NY

```
console . log ( User . magic_number ( 4 ));
```

Output: 16

User is the object.

to access any element of the User object  
we use dot operator i.e. User.name

## Strings

```
let sensText = "thisIsSesnText";  
let maskedText =  
sensText.substring(3).padStart(sensText.le  
ngth, "*");  
console.log( maskedText)
```

Output: \*\*\*sIsSesnText

code to delete a key and value pair

```
let User = {
```

```
    name: 'John',
```

```
    age: 30,
```

```
    phone: '123-456-7890',
```

```
    address: {
```

```
        street: '123 Main St',
```

```
        city: 'New York',
```

```
        state: 'NY',
```

```
        zip: '10001'
```

```
},
```

```
hobbies: ['movies', 'music'],
```

```
magic_number: function (x) {
```

```
    return x * x;
```

```
}
```

```
};
```

```
console.log(User.age);
```

```
delete User.age; // delete age
```

```
console.log(User.age);
```

Output:

30

undefined

# Arrays

Similar to vector in C++.

Can hold anything inside the array.

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];  
console.log(arr);
```

Output:

```
[  
 1, 2, 3, 4,  
 5, 6, 7, 8  
]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];  
arr.push(123); // push at the end  
console.log(arr);
```

Output:

```
[
```

```
1, 2, 3, 4, 5,  
6, 7, 8, 123  
]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];  
arr.pop(); // pops last element and returns  
it
```

```
console.log(arr);
```

Output:

```
[  
 1, 2, 3, 4,  
 5, 6, 7  
]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];  
let lastEle = arr.pop();  
console.log(arr);  
console.log(" ",lastEle);
```

Output:

```
[  
 1, 2, 3, 4,
```

5, 6, 7

]

8

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
let lastEle = arr.shift(); // pops element at
the beginning and returns it
console.log(arr)
console.log("  ",lastEle);
```

Output:

[

2, 3, 4, 5,

6, 7, 8

]

1

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
let lastEle = arr.unshift(); // inserts the
given value at the beginning of the array
and returns the length of the new array
console.log(arr)
```

```
console.log(" ",lastEle);
```

Output:

```
[  
 1, 2, 3, 4,  
 5, 6, 7, 8  
]  
8
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];  
let lastEle = arr.unshift(1);  
console.log(arr)  
console.log(" ",lastEle);
```

Output:

```
[  
 1, 1, 2, 3, 4,  
 5, 6, 7, 8  
]  
9
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];  
let slicedArr = arr.slice(1,4) // to make
```

subarrays of the given array with particular elements

```
console.log(slicedArr)
```

Output:

```
[ 2, 3, 4 ]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
```

```
let removedArr = arr.splice(1,2) // to  
remove particular elements from a  
particular index and or also to add index  
after the index
```

```
console.log(arr)
```

Output:

```
[ 1, 4, 5, 6, 7, 8 ]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
```

```
let removedArr = arr.splice(1,2,99)
```

```
console.log(arr)
```

Output:

```
[
```

```
 1, 99, 4, 5,
```

```
 6, 7, 8
```

]

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
let removedArr = arr.splice(1,2,99,88,77)
console.log(arr)
```

Output:

```
[  
  1, 99, 88, 77, 4,  
  5, 6, 7, 8  
]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
let removedArr = arr.splice(1,2,[99,88,77])
console.log(arr)
```

Output:

```
[ 1, [ 99, 88, 77 ], 4, 5, 6, 7, 8 ]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
let removedArr = arr.splice(1,2,
[99,88,77],'test')
```

```
console.log(arr)
```

Output:

```
[ 1, [ 99, 88, 77 ], 'test', 4, 5, 6, 7, 8 ]
```

```
let arr1 = [1, 2, 3, 4, 5, 6, 7, 8];
```

```
let arr2 = [22,33,44];
```

```
let arr3 = arr1.concat(arr2); // combine  
two arrays
```

```
console.log(arr3);
```

Output:

```
[
```

```
 1, 2, 3, 4, 5,
```

```
 6, 7, 8, 22, 33,
```

```
44
```

```
]
```

```
let arr = [1, 2, 3];
```

```
let outputStr = arr.join("=")
```

```
console.log(outputStr);
```

Output:

```
1=2=3
```

```
let arr = [1, 2, 3];
let outputStr = arr.join("= | =") // converts
array to string with a given delimiter
console.log(outputStr);
```

Output:

```
1= | =2= | =3
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
console.log(arr.indexOf(3)); // returns
index of an element and if not found return
-1
```

Output:

```
2
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
arr.forEach( (e) => {
    console.log(e);
}); // access each element and print it
```

## Output:

1  
2  
3  
4  
5  
6  
7

let arr = [1, 2, 3, 4, 5, 6, 7]

let filtArr = arr.filter((e) => e%2) //

segregates values on the basis of truth or  
false values

passes every element through the function  
and adds to the array if the value is true  
else discards the value after performing  
the specified operation.

console.log(filtArr)

## Output:

[ 1, 3, 5, 7 ]

```
let arr = [1, 2, 3, 4, 5, 6, 7]
let filtArr = arr.filter((e) => e < 3)
console.log(filtArr)
```

Output:

```
[ 1, 2 ]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
let filtArr = arr.filter((e) => e > 3)
console.log(filtArr)
```

Output:

```
[ 4, 5, 6, 7 ]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
let filtArr = arr.filter((e) => e % 2 === 1)
console.log(filtArr)
```

Output:

```
[
  1, 2, 3, 4,
  5, 6, 7
]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
let filtArr = arr.filter((e) => e > 0)
console.log(filtArr)
```

Output:

```
[]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
let mappedArr = arr.map((e) => e + 3); //  
passes every element through the function  
and returns the value after performing the  
specified operation.
console.log(mappedArr);
```

Output:

```
[  
  4, 5, 6, 7,  
  8, 9, 10  
]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
let mappedArr = arr.map((e) => e * 2);
console.log(mappedArr);
```

Output:

```
[  
 2, 2, 2, 2,  
 2, 2, 2  
]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
```

```
let mappedArr = arr.map((e) => `${e} abc`);  
console.log(mappedArr);
```

Output:

```
[  
 '1 abc', '2 abc',  
 '3 abc', '4 abc',  
 '5 abc', '6 abc',  
 '7 abc'  
]
```

```
let arr = [1, 2, 3, 4, 5, 6, 7]
```

```
let mappedArr = arr.map((e) => `${e} is a  
number`);
```

```
console.log(mappedArr);
```

Output:

```
[  
  '1 fis a number',  
  '2 fis a number',  
  '3 fis a number',  
  '4 fis a number',  
  '5 fis a number',  
  '6 fis a number',  
  '7 fis a number'  
]
```

```
let arr = [1, 2, 3]  
let sum = arr.reduce((acc, num) => acc +  
num, 10); // here the accumulater is  
initialized with a value and then the value  
gets added with the accumulator untill all  
elements are accessed by the num  
console.log(sum);
```

Output:

16

## Array Operations:

- `push` --> add element at the end.
- `pop` --> remove element at beginning.
- `shift` --> deletes 1st element and returns it.
- `unshift` --> add element at the beginning and returns the size of the new array.
- `slice` --> create subarrays with elements in a range which includes the 1st element and not the last element i.e., `slice(1,4)` includes element at `index=1` and doesn't include element at `index=4`.
- `splice` --> remove particular elements from a particular index and also to add index after the index
- `concat` --> combining to arrays
- `join` --> add some string between the

array elements while printing

- `indexOf` --> returns index of an element
- `forEach` --> access each element and perform an operation as specified
- `filter` --> segregates values on the basis of truth or false values
- `map` --> passes every element through the function and returns the value after performing the specified operation.

## Nested Ternary Operator

```
let user = {  
    age: 17,  
    role: 'admin',  
};
```

```
let eligible = user.age > 18 ? true : user.role  
==== 'admin' ? true : false;  
console.log(eligible);
```

Output:

true

why not to use this? readability takes a toll

## Using a predefined function in map

```
let arr = [1, 2, 3, 4, 5, 6, 7]
let mappedArr = arr.map((e) => process(e));
console.log(mappedArr);
function process(k)
{
    return `Processed value is ${k+2}`;
}
```

Output:

```
[  
'Processed value is 3',  
'Processed value is 4',  
'Processed value is 5',  
'Processed value is 6',  
'Processed value is 7',  
'Processed value is 8',  
'Processed value is 9'  
]
```

```
let arr =
```

```
["RA20123","RA20127","RA20126","RA20123  
4"];  
let mappedArr = arr.map((e) => process(e));  
console.log(mappedArr);  
function process(k) {  
    // READ ROM DB  
    // VALIDATE  
    // RETURN  
    return k;  
}
```

```
function getResponse(){  
    let addr = "https://  
jsonplaceholder.typicode.com/todos/1";  
    // this has not yet been implemented  
    const response = fetch(addr);  
    const data = response.json();  
    console.log(data);  
}  
getResponse();
```

**Output:**

**Error**

```
// not working because fetch is  
asynchronous. As the response is not yet  
received, the data is undefined.
```

```
// event loop --> main thread on which  
javascript runs. It is a single threaded  
language.
```

```
setTimeout(() => {
  console.log('Hellow!');
}, 4000);
```

```
console.log('Yes');
setTimeout(() => {
  console.log('Hellowww!');
}, 2000);
```

// executed based on the time it takes to execute the code

## Using async-await function to get response from the url

async --> to define that the process will take a undefined amount of time  
await --> waits in the async until the value is completely fetched

```
async function getResponse(){
  let addr = "https://
```

```
jsonplaceholder.typicode.com/todos/1";  
    const response = await fetch(addr);  
    const data = await response.json();  
    console.log(data);  
}  
getResponse();
```

Output:

```
{ userId: 1, id: 1, title: 'delectus aut autem',  
completed: false }
```

```
async function getResponse(){  
    let addr = "https://  
jsonplaceholder.typicode.com/todos/1";  
    const response = await fetch(addr);  
    const data = response.json();  
    console.log(data);  
}  
getResponse();
```

Output:

Promise { <pending> }

Promise states:

- pending
- resolved
- rejected

Pre-historic version of async-await

```
const displayDelay = new
Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Met at 06:30");
  }, 1000);
});
displayDelay.then((v)=> {
  console.log("SUCCESS: ", v);
});
```

Output:

SUCCESS: Met at 06:30

```
const displayDelay = new  
Promise((resolve, reject) => {  
    setTimeout(() => {  
        reject("Met at 06:30");  
    }, 1000);  
});  
displayDelay.then((v)=> {  
    console.log("SUCCESS: ", v);  
});
```

Output:

```
node:internal/process/promises:288  
    triggerUncaughtException(err, true /  
* fromPromise */);  
    ^
```

[UnhandledPromiseRejection: This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). The promise rejected with the reason "Met at 06:30".] {  
code: 'ERR\_UNHANDLED\_REJECTION'

}

Here we didn't write code for what happens when there is error or it gets rejected hence got the error.

So in the below code this error has been removed to get the output by defining catch function which performs the respective task when the function is executed and gets error.

```
const displayDelay = new
Promise((resolve, reject) => {
  setTimeout(() => {
    reject("Met at 06:30");
  }, 1000);
});
displayDelay
  .then((v)=> {
    console.log("SUCCESS: ", v);
})
  .catch((e) => {
```

```
    console.log("ERROR: ", e);
});
```

Output:

```
ERROR: Met at 06:30
```

Due to callback hell we moved from promise to async-await.

To know why we moved from the above code to async-await.

ATM machines are the examples of async-await.

```
const displayDelay = new
Promise((resolve, reject) => {
// if user found in database:
    // resolve("User found");
// if user not found in database:
    // reject("User not found");
});
```

```
displayDelay
  .then((v)=> {
    // generateCertificate
  })
  .catch((e) => {
    // displayError
  });
});
```

Promise are used to handle uncertainties.

```
const displayDelay = new
Promise((resolve, reject) => {
// if user found in database:
  // resolve("User found");
// if user not found in database:
  // reject("User not found");
});
displayDelay
  .then((v)=> {
    // generateCertificate
  })
```

```
.catch((e) => {
  // displayError
});
```

The above code is for try and catch in async-await. Basically if we get error it can be resolved.

```
async function getResponse(){
  let addr = "https://
jsonplaceholder.typicode.com/todos/1";
  try {
    const response = await fetch(addr);
    const data = await response.json();
    console.log(data);
  } catch (e) {
    console.log(e);
  }
}
getResponse();
```

Output:

```
{ userId: 1, id: 1, title: 'delectus aut autem',  
completed: false }
```

What is an API?

Rest API:

Representational State Transfer Interface

input --> object

output --> mostly object

npm --> collection of packages made by  
developers

Installing Express:

npm init // entry point: (x.js) index.js

npm i express

Download Postman agent

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`)
})
```

Output:

Example app listening on port 3000

## Postman Output:

The screenshot shows the Postman interface with a successful API call. The URL is set to `localhost:3000/`. The request method is `GET`, and the response status is `200 OK`. The response body is displayed as "Hello World!".

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send({hello: "world"})
})

app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`)
})
```

## Postman Output:

The screenshot shows the Postman interface with a successful API call. The URL is set to `localhost:3000/`. The request method is `GET`. In the `Body` tab, the raw JSON payload sent is:

```
1 | {
2 |   "name": "Blessin"
3 | }
```

The response status is `200 OK`, with a time of `25 ms` and a size of `252 B`. The response body is displayed as:

```
1 | {
2 |   "hello": "world"
3 | }
```

```
const express = require('express')
const app = express()
app.use(express.json());
const port = 3000

app.get('/', (req, res) => {
  res.send({hello: "world from get"})
})
app.post('/', (req, res) => {
  res.send({hello: "world from post"})
})
app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`)
})
```

Postman Output:

localhost:3000/

localhost:3000/

GET localhost:3000/

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Beautify

• none • form-data • x-www-form-urlencoded • raw • binary • GraphQL JSON

```
1: {  
2:   "name": "Hellozin"  
3: }
```

Body Cookies Headers (7) Test Results Status: 200 OK Time: 37 ms Size: 261 B Save Response

Pretty Raw Preview Visualize JSON

```
1: {  
2:   "Hello": "world From get"  
3: }
```

localhost:3000/

localhost:3000/

POST localhost:3000/

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Beautify

• none • form-data • x-www-form-urlencoded • raw • binary • GraphQL JSON

```
1: {  
2:   "name": "Hellozin"  
3: }
```

Body Cookies Headers (7) Test Results Status: 200 OK Time: 9 ms Size: 262 B Save Response

Pretty Raw Preview Visualize JSON

```
1: {  
2:   "Hello": "world from post"  
3: }
```

```
const express = require('express')
const app = express()
app.use(express.json());
const port = 3000

app.get('/', (req, res) => {
  res.send({hello: "world from get"})
})
app.post('/', (req, res) => {
  console.log(req.body.name)
  res.send({hello: "world from post"})
})
app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`)
})
```

Postman Output:

localhost:3000/

localhost:3000/

GET

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Beautify

Body

```
1 [ ]  
2 { "name": "Blessin"  
3 }
```

Body Cookies Headers (7) Test Results

Status: 200 OK Time: 37 ms Size: 261 B Save Response

localhost:3000/

localhost:3000/

POST

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Beautify

Body

```
1 [ ]  
2 { "name": "Blessin"  
3 }
```

Body Cookies Headers (7) Test Results

Status: 200 OK Time: 9 ms Size: 262 B Save Response

```
PS G:\Node Js and React Js> node "g:\Node Js and React Js\index.js"
Example app listening on port 3000
Blessin
```

```
const express = require('express')
const app = express()
app.use(express.json());
const port = 3000

app.get('/', (req, res) => {
  res.send({hello: "world from get"})
})
app.post('/', (req, res) => {
  console.log(req.body)
  res.send({hello: "world from post"})
})
app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`)
})
```

Postman Output:

The screenshot shows the Postman application interface with two requests:

- GET Request:** URL: `localhost:3000/`.
  - Body tab (selected):

```
1 [ {  
2   "name": "Blessin"  
3 } ]
```
  - Response status: 200 OK, Time: 37 ms, Size: 261 B.
- POST Request:** URL: `localhost:3000/`.
  - Body tab (selected):

```
1 {  
2   "name": "Blessin"  
3 }
```
  - Response status: 200 OK, Time: 9 ms, Size: 262 B.

```
PS G:\Node Js and React Js> node "g:\Node Js and React Js\index.js"
Example app listening on port 3000
{ name: 'Blessin' }
```

```
const express = require('express')
const app = express()
app.use(express.json());
const port = 3000

app.get('/', (req, res) => {
  res.send({hello: "world from get"})
})
app.post('/', (req, res) => {
  let user = req.body.name;
  res.send({name: user})
})
app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`)
})
```

Postman Output:

The screenshot shows two requests made using the Postman application.

**Request 1: GET localhost:3000/**

- Method:** GET
- URL:** localhost:3000/
- Body (JSON):**

```
1 {
2   "name": "Ulessin"
3 }
```

- Response Status:** 200 OK
- Response Time:** 8 ms
- Response Size:** 261 B

**Request 2: POST localhost:3000/**

- Method:** POST
- URL:** localhost:3000/
- Body (JSON):**

```
1 {
2   "name": "Ulessin"
3 }
```

- Response Status:** 200 OK
- Response Time:** 37 ms
- Response Size:** 253 B

Put:

Patch:

## Installing nodemon:

it is used for faster real time work in javascript

`npm i nodemon -g`

To run the file

`nodemon index.js`

if not working:

open powershell as admin

run the below command

- `Set-ExecutionPolicy RemoteSigned -Scope CurrentUser`

```
const express = require('express')
const app = express()
app.use(express.json());
const port = 3000

app.get('/', (req, res) => {
  res.send({hello: "world from get api"})
})

app.post('/', (req, res) => {
  let user = req.body.name;
  res.send({name: user})
})

app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`)
})
```

Ouput:

When the code is run gets updated as we run the send button in postman. Hence automated the manual process of running

in terminal again and again.

```
const express = require("express");
const app = express();
app.use(express.json());
const port = 3000;

app.get("/", (req, res) => {
  res.send({ hello: "world from get api endpoint" });
});
app.post("/", (req, res) => {
  let user = req.body.name;
  res.send({ name: user });
});
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```

```
// in terminal:  
// npm i nodemon -g  
// nodemon index.js
```

```
const express = require("express");  
const app = express();  
app.use(express.json());  
const port = 3000;  
  
app.get("/", (req, res) => {  
    res.send({ hello: "world from get api  
endpoint" });  
});  
app.post("/", (req, res) => {  
    let a = req.body.a;  
    let b = req.body.b;  
    let c = a + b;  
    res.send({ result: c });  
});
```

```
app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`);
});
```

## Output:

The screenshot shows two separate Postman requests to the endpoint `localhost:3000/`.

**Request 1 (Top):**

- Method: POST
- Body type: JSON
- Body content:

```
1: {  
2:   "a": 6,  
3:   "b": 5  
4: }
```
- Response status: 200 OK
- Response body:

```
1: {}  
2:   "result": 11  
3: {}
```

**Request 2 (Bottom):**

- Method: POST
- Body type: JSON
- Body content:

```
1: {  
2:   "a": "888",  
3:   "b": 6  
4: }
```
- Response status: 200 OK
- Response body:

```
1: {}  
2:   "result": "888"  
3: {}
```

The screenshot shows the Postman application interface. At the top, it says "localhost:3000" and "No Environment". Below that, the URL "localhost:3000/" is entered. The method is set to "POST". The "Body" tab is selected, showing a JSON object with one field: "result": null. The "Pretty" tab is selected in the preview area. At the bottom right, the status is shown as "Status: 200 OK Time: 5 ms Size: 240 B".

```
const express = require("express");
const app = express();
app.use(express.json());
const port = 3000;
```

```
app.get("/", (req, res) => {
  res.send({ hello: "world from get api endpoint" });
});
app.post("/", (req, res) => {
  let a = req.body.a;
```

```
let b = req.body.b;
let op = req.body.op;
let c = 0;
switch(op)
{
  case '+':
    c = a + b;
    break;
  case '-':
    c = a - b;
    break;
  case '*':
    c = a * b;
    break;
  case '/':
    c = a / b;
    break;
}
res.send({ result: c });
});
```

```
app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`);
});
```

The screenshot shows the Postman application interface. At the top, it says "POST localhost:3000/" and "localhost:3000/". Below that, there's a "Send" button and some other UI elements. The main area has tabs for "Params", "Authorization", "Headers (0)", "Body", "Pre-request Script", "Tests", and "Settings". The "Body" tab is selected and contains the following JSON:

```
1
2   {
3     "a": 12,
4     "b": 6,
5     "op": "+"
6 }
```

Below the body, there's a "Cookies" section and a "Beautify" button. At the bottom, there are tabs for "Body", "Cookies", "Headers (0)", and "Test Results". The "Test Results" tab is selected and shows the following JSON response:

```
1
2   {
3     "result": 18
4 }
```

At the very bottom, there are buttons for "Pretty", "Raw", "Preview", "Visualize", "JSON", and "Save Response".

Using a separately defined function:

```
const express = require("express");
const app = express();
app.use(express.json());
const port = 3000;
```

```
app.get("/", (req, res) => {
```

```
res.send({ hello: "world from get api endpoint" });
});

app.post("/", (req, res) => {
  let a = req.body.a;
  let b = req.body.b;
  let op = req.body.op;
  let c = 0;
  c = operation(a,b,op)
  res.send({ result: c });
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});

function operation(a,b,op){
  switch(op)
  {
    case '+':
      return a + b;
    case '-':

```

```
    return a - b;  
    case '*':  
        return a * b;  
    case '/':  
        return a / b;  
    }  
}  
}
```

## Output:

The screenshot shows the Postman interface with a POST request to `localhost:3000/`. The request body is a JSON object with fields `a`, `b`, and `op`. The response status is 200 OK, and the response body is a JSON object with a single field `result1`.

Request Body:

```
1 {  
2     "a": 12,  
3     "b": 6,  
4     "op": "/"  
5 }
```

Response Body:

```
1 {  
2     "result1": 2  
3 }
```

localhost:3000/?a=4&b=6

url query parameter

```
const express = require("express");
const app = express();
app.use(express.json());
const port = 3000;

app.get("/", (req, res) => {
  res.send({ hello: "world from get api
endpoint" });
});
app.post("/", (req, res) => {
  let a = parseInt(req.query.a);
  let b = parseInt(req.query.b);
  let c = a + b;
  res.send({ result: c });
});
app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`);
```

```
port ${port}`);
```

```
});
```

## Output:

The screenshot shows the Postman application interface. At the top, it displays the URL `localhost:3000/?a=4&b=8`. Below this, the method is set to `POST` and the endpoint is `localhost:3000/?a=4&b=8`. The `Body` tab is selected, showing the message "This request does not have a body". In the bottom section, under the `Body` tab, the response is displayed in JSON format. The response body is a single key-value pair: `"result": 18`.

```
const express = require("express");
```

```
const app = express();
```

```
app.use(express.json());
```

```
const port = 3000;
```

```
app.get("/", (req, res) => {
```

```
    res.send({ hello: "world from get api"
```

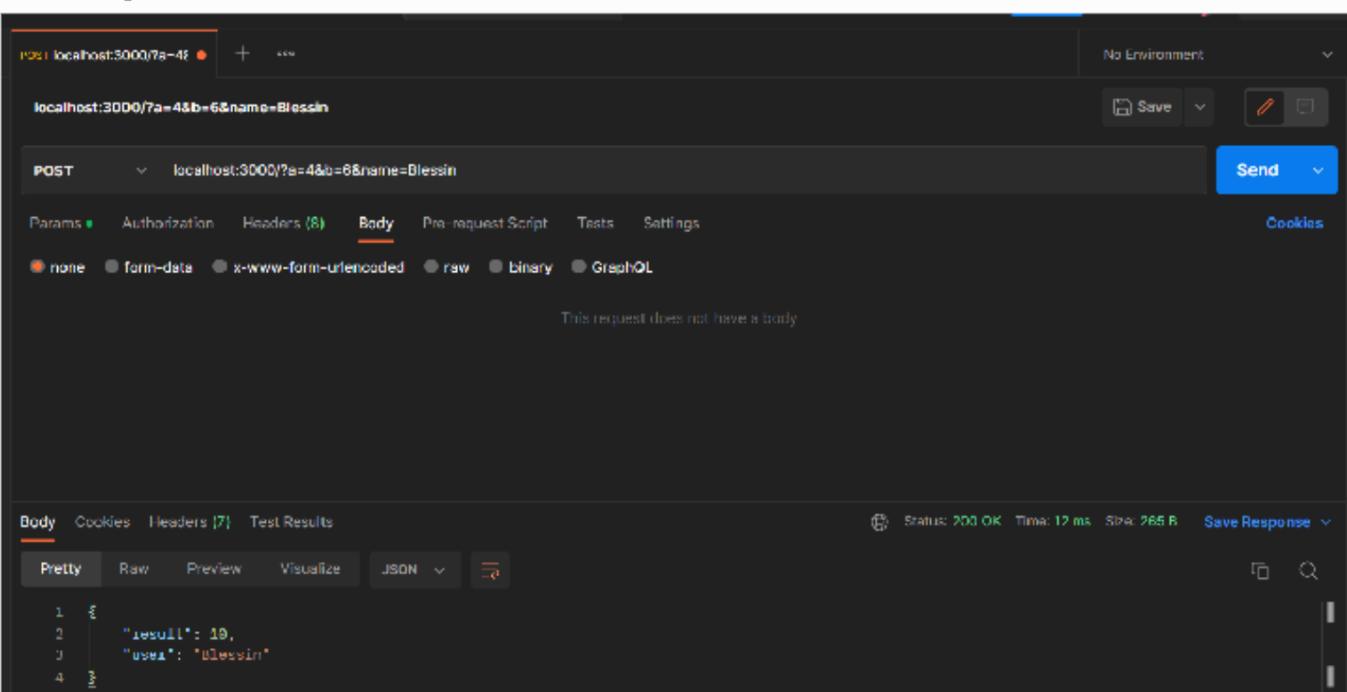
```
endpoint" });

});

app.post("/", (req, res) => {
  let a = parseInt(req.query.a);
  let b = parseInt(req.query.b);
  let name = req.query.name;
  let c = a + b;
  res.send({ result: c, user: name });
});

app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`);
});
```

## Output:



The screenshot shows the Postman application interface. At the top, the URL is set to `localhost:3000/?a=4&b=6&name=Blessin`. Below the URL, the method is selected as `POST` and the endpoint is `localhost:3000/?a=4&b=6&name=Blessin`. The `Body` tab is active, showing the message "This request does not have a body". In the bottom section, the `Body` tab is selected, displaying the response in JSON format:

```
1  {
2   "result": 10,
3   "user": "Blessin"
4 }
```

The status bar at the bottom indicates a `Status: 200 OK`, `Time: 12 ms`, and `Size: 266 B`.

# Now using "freedb.tech" website to create database.

The screenshot shows the Freedb Panel interface. On the left, there's a sidebar with icons for Dashboard, PRO PANEL, USER PROFILE, ANSWERS, FAQ, and SERVER STATISTICS. The main area has a 'Welcome To FreeDB V2' message with the Freedb logo. On the right, there's a 'FreeDB Free Database Panel' section with fields for HOST (localhost), PORT (3306), DATABASE NAME (test\_db\_140824), and a database user entry (freedb\_Ms\_23-31313) with password UP240547Lc&n@2n. Below these are checkboxes for MySQL 8.0 and PhpMyAdmin 4.1.

## Installing mysql: npm i mysql

## Now creating a new folder db creating config.js file in db

## config.js file

```
const mysql = require('mysql2');
const connection =
mysql.createConnection(
{
```

```
host: 'sql.freedb.tech',
  user: 'freedb_Me_23131313',
  password: 'UP2#C5#7uG&n@Zh',
  database: 'freedb_Test_Database'
});

connection.connect((err)=>{
  if(err)
  {
    console.log(err);
  }
  else
  {
    console.log('Connected to database');
  }
});
```

Run: node db/config.js

```
PS G:\Node Js and React Js> node db/config.js
Debugger attached.
Connected to database
```

Output:

Now open website "phpmyadmin.co"  
fill the details from freedb.tech  
click on new  
create table with name as 'users' and  
column name as 5

The screenshot shows the phpMyAdmin interface for a database named 'freedb'. On the left, the database structure is visible with tables like 'friends', 'friends\_requests', and 'users'. The 'users' table is currently selected. The 'Structure' tab is active, displaying the table's columns:

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int	utf8mb4_unicode_ci	Not Null	Yes			AUTO_INCREMENT	
2	name	varchar(50)	utf8mb4_unicode_ci	Not Null	Yes				
3	phone	varchar(15)	utf8mb4_unicode_ci	Not Null	Yes				
4	role	varchar(50)	utf8mb4_unicode_ci	Not Null	Yes				
5	enabled	int		Not Null	Yes				

Below the table structure, there are tabs for 'Indexes' and 'Collisions'. The 'Indexes' tab shows a single index entry:

Action	Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
	PRIMARY	MUL	Yes	No	id	1	A	No	

The 'Collisions' tab is empty.

At the bottom, the SQL tab contains the generated SQL code:

```
CREATE TABLE `friends`(`id` int NOT NULL AUTO_INCREMENT,`name` varchar(50) NOT NULL,`phone` varchar(15) NOT NULL,`role` varchar(50) NOT NULL,`enabled` int NOT NULL,`created` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,`updated` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,`deleted` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00') ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

# What are the different ways to track a customer?

Cookie

JWT --> Json Web Tokens

## Config file for xampp

## Config file for xampp

The screenshot shows the phpMyAdmin interface for a MySQL database named 'newdb'. The left sidebar shows the database structure with a single table called 'customers'. The main area displays the 'customers' table with the following data:

ID	name	email	role	enabled
1	John Doe	john.doe@example.com	customer	1
2	Sarah Smith	sarah.smith@example.com	customer	1
3	David Lee	daavid.lee@example.com	customer	1
4	Emily Green	emily.green@example.com	customer	0
5	Michael White	michael.white@example.com	admin	1

Below the table, there are buttons for 'Check', 'View details', 'Edit', 'Delete', 'Import', and 'Export'.

```
const mysql = require('mysql2');
const connection =
mysql.createConnection(
{
  host: 'sql.freedb.tech',
  user: 'freedb_weuidfjwe841',
  password: 'bStEB%88Ac%zs@*',
  database: 'freedb_nowbegin123'
});
connection.connect((err)=>{
  if(err)
  {
    console.log(err);
  }
  else
  {
    console.log('Connected to database');
  }
});
module.exports = connection;
```

Unopioniated --> where we can choose names of the folders in the directory.

Opinionated --> where we can't choose named of the folders in the directory.

request is generated and it hits express than if the "/users" is identified than if the router is identified than the request is given to the router than it is detects "/" than passes the access to the controller file.

router --> similar to a sign board controller file is used to access the database

404 --> the route does not exist.

# INDEX.JS

```
const express = require("express");
const app = express();
const userRoutes = require("./routes/
user.routes");

app.use(express.json());
const port = 3000;

app.use("/users",userRoutes);

app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`);
});
```

# USER.ROUTES.JS

```
const express = require('express');
const userController = require('../
controllers/user.controller');
const router = express.Router();

router.get("/",userController.getAllUsers);

module.exports = router;
```

# USER.CONTROLLER.JS

```
const connection = require("../db/config");
module.exports = {
  getAllUsers:(req, res)=>{
    connection.query("SELECT * FROM
users", (err, results)=>{
      if(err) {
```

```
        res.status(500).send("Error
retrieving users");
    }
    else
    {
        res.json(results);
    }
});
```

}

}

## Output:

The screenshot shows the Postman application interface. At the top, the URL bar displays "localhost:3000/users/" and the status "No Environment". Below the URL bar, the request type is set to "GET" and the endpoint is "localhost:3000/users/". The "Headers" tab is selected, showing a single header "Content-Type: application/json". The "Body" tab is also present. In the main body area, it says "This request does not have a body". At the bottom, the response section is visible with the status "Status: 200 OK", time "Time: 332 ms", size "Size: 807 B", and a "Save Response" button. The "Pretty" tab is selected in the response body, displaying the JSON data:

```
1 [ 
2   {
3     "id": 1,
4     "name": "rahon",
5     "email": "Test@sis.sen",
6     "role": "Web dev",
7     "enabled": 1
8   },
9 ]
```

# Index.js

```
const express = require("express");
const app = express();
const userRoutes = require("./routes/
user.routes");
app.use(express.json());
const port = 3000;

app.use("/users",userRoutes);

app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`);
});
```

# User.routes.js

```
const express = require('express');
const userController = require('../
controllers/user.controller');
const router = express.Router();

router.get("/",userController.getAllUsers);
router.get("/:id",userController.getUserByID)
;

module.exports = router;
```

# User.controller.js

```
const connection = require("../db/config");
module.exports = {
  getAllUsers:(req, res)=>{
    connection.query("SELECT * FROM
users", (err, results)=>{
      if(err) {
```

```
        res.status(500).send("Error
retrieving users");
    }
    else
    {
        res.json(results);
    }
});
},
getUserByID:(req, res)=>{
    connection.query(`SELECT * FROM
users WHERE id = ${req.params.id}`, (err,
results)=>{
    if(err) {
        res.status(500).send("Error
retrieving users");
    }
    else
    {
        res.json(results);
    }
})
```

```
});  
}  
}  
}
```

## Output:

The screenshot shows the Postman application interface. At the top, the URL is set to `localhost:3000/users/3`. Below the URL, the method is selected as `GET`. The `Body` tab is active, showing the message: "This request does not have a body". In the `Test Results` section, the response status is `200 OK`, time is `289 ms`, and size is `305 B`. The response body is displayed in Pretty, Raw, Preview, and Visualize formats, all showing the same JSON object:

```
1  {  
2   "id": 3,  
3   "name": "boy",  
4   "email": "boy@buy.com",  
5   "role": "boy",  
6   "emailed": 1  
7 }  
8  
9
```

# Using SQL WORKBENCH for database

## Index.js

```
const express = require("express");
const app = express();
const userRoutes = require("./routes/
user.routes");

app.use(express.json());
const port = 3000;

app.use("/users",userRoutes);

app.listen(port, () => {
  console.log(`Example app listening on
port ${port}`);
});
```

## User.routes.js

```
const express = require('express');
const userController = require('../
controllers/user.controller');
const router = express.Router();

router.get("/",userController.getAllUsers);

router.get("/:id",userController.getUserByID)
;

module.exports = router;
```

## User.controller.js

```
const connection = require("../db/config");

module.exports = {
  getAllUsers:(req, res)=>{
    connection.query("SELECT * FROM episodes", (err, results)=>{
      if(err) {
        res.status(500).send("Error retrieving users");
      }
      else
      {
        res.json(results);
      }
    });
  },
  getUserByID:(req, res)=>{
    connection.query(`SELECT * FROM episodes WHERE id = ${req.params.id}`,
```

```
(err, results)=>{
    if(err) {
        res.status(500).send("Error
retrieving users");
    }
    else
    {
        res.json(results);
    }
});
```

}

## Config.js

```
const mysql = require('mysql2');
const connection =
mysql.createConnection(
{
    host: 'localhost',
```

```
    user: 'root',
    password: '12345',
    database: 'anime'

  });

connection.connect((err)=>{
  if(err)
  {
    console.log(err);
  }
  else
  {
    console.log('Connected to database');
  }
});

module.exports = connection;
```

# Errors

200 success

300 redirected

500 server error

403 forbiden



