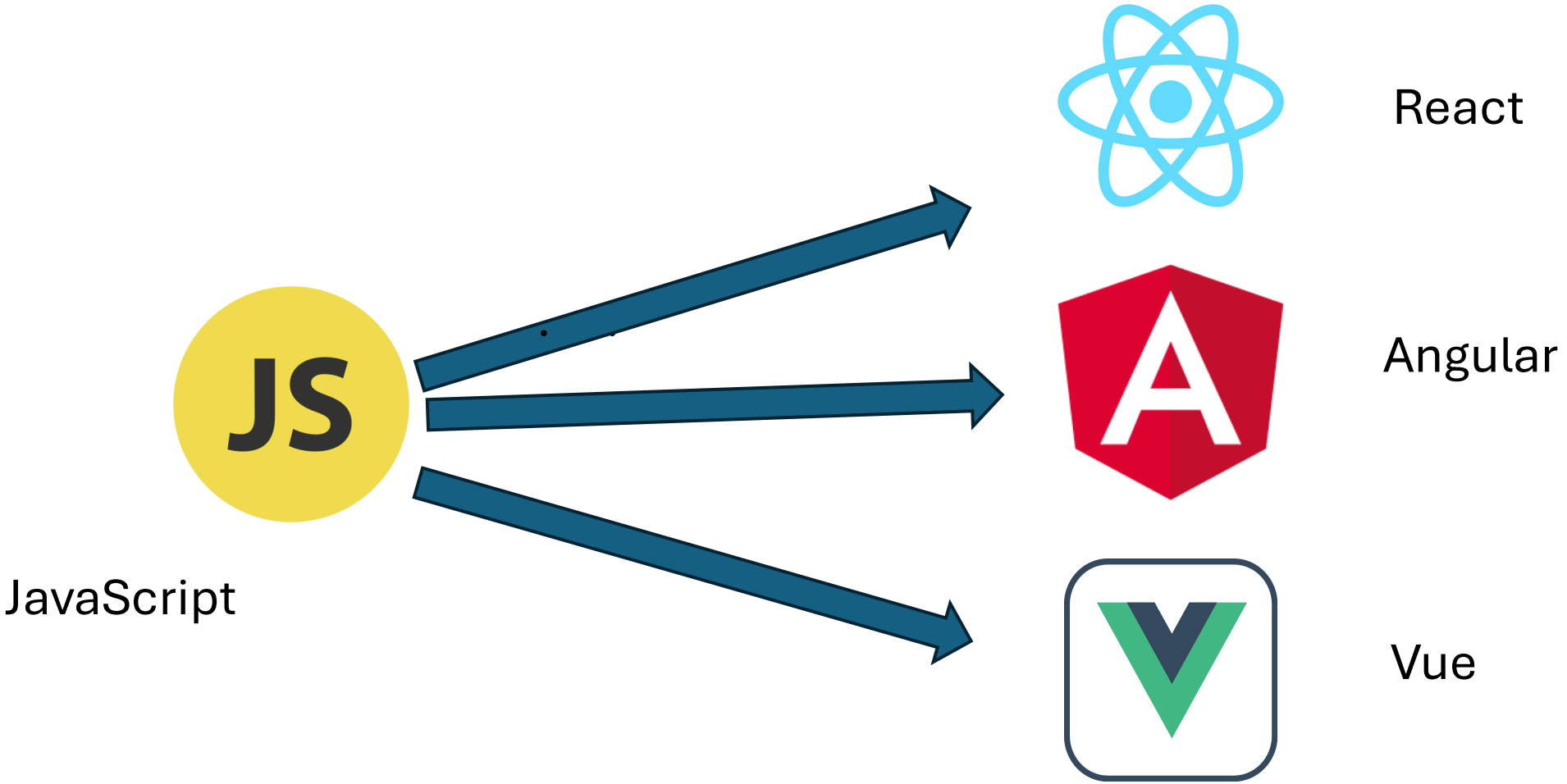
JavaScript Introduction

What can I do with JavaScript

- Front-End
- Back-End
- Mobile
- Desktop
- Etc....

Front-End JavaScript



Back-End JavaScript

Nodejs allows JavaScript to be run outside the browser, i.e. on a server

Late on, we'll use Node.js to create a simple REST API

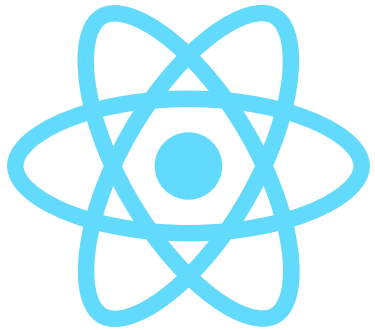
Back-end JavaScript

Allows developers to publish and share code, making development a lot faster

Back-End JavaScript

A non-relational database that stores JavaScript objects

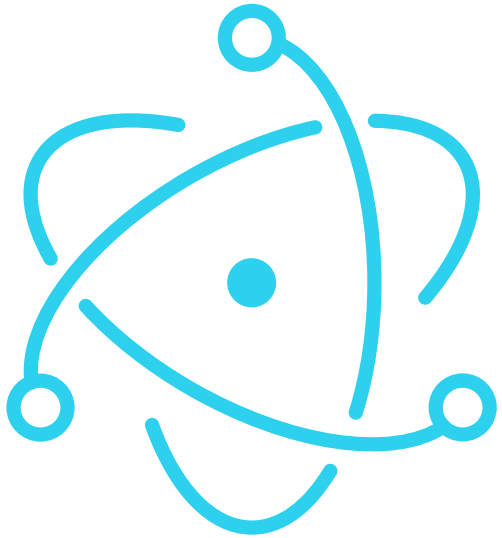
JavaScript for Mobile



Allow us to write mobile apps with JavaScript

One of the more recent additions to the JavaScript universe are libraries like React Native and NativeScript which give us the ability to write mobile apps with JavaScript. it means that instead of having to learn Swift and Java to write mobile apps for iPhone and Android, we can simply write them using JavaScript. We can have a single central code base that contains all of the front-end logic for our applications. A single team of JavaScript developers can create and maintain a multi-platform app where it used to take a JavaScript dev team, an iPhone dev team, and an Android dev team to do the same amount of work.

JavaScript for Desktop



Electron allows us to build desktop apps using JavaScript

Electron takes the front-end apps you've written in JavaScript and builds them so that they can be run as a desktop application. Probably the most popular desktop app that's built with Electron is Slack.

JavaScript Language Features

- JavaScript is NOT Java
- JavaScript is an Interpreted language
  Excetued without compiling first
  The JavaScript runtime reads actual JavaScript code, not bytecode.
  The exception to this is JIT compilation for performance reason.
- JavaScript is dynamically and weakly typed language
  Variable types determinated at runtime not compile time.
  Opposite is "static typing"
  Let x= 5 vs int x =5
- "Mostly" Object-Oriented
- JavaScript is Single-threaded
  Processing large amounts of data isn't one of JavaScript's strong suits.

The "Pros" of JavaScript

- It's incredibly popular
- It's relatively easy to learn and use
- Can be used for a very wide range of applications
- Code can be executed on the client side, which reduces the load on the server

The "Cons" of JavaScript

- We have to worry about browser support
- We have to be more security-conscious

  The client has a copy of our source code.
- Certain parts of the language can be behave very strangely
- Not the best language for OOP
- As a language, JavaScript is evolving very quickly

JavaScript DataType

Defining Variables and Constants

Function-scoped

```
var  x=5
```

Block-Scoped

```
let s = 'hello'
```

```
const PI=3.14
 PI=3    // -> Error
```

```
globalVariable = "GLBOAL"
```

The 8 Data Types

- Numbers
- Strings
- Booleans
- Objects
- Functions
- Undefined
- Bigints
- Sysbols

Variables don't have Types

let x =5;   // x is a number

x= "String" ; // x is a string.

Variables don't have types, only values have types

x = 5
typeof x;   => "number"

Number type

```
let x = 5;

let pi=3.14
let x = 0x2A
let binary = 0b1010
let octal = 0o52

let sixtyMillion = 6e7
"NaN"     let answer = 10 *"oops"       // => NaN
"Infinity"   let answer = 10/0   // => Infinity
                  let answer = Math.pow(10, 9999);   // => Infinity
```

Number Precision

- All number values are stored as 64-bit floating point
  Big numbers are only accurate to 15 digits

  let x = 0.1 + 0.7
- If(x === 0.8) {
    // might not execute!
- }
  if(Math.round(x*10) === Math.round(0.8*10){

// this works!
}

The String Type

```
let singleQuoteString = 'Hello'
let doubleQuoteString = "Hello"
let backtickString = `Hello ${name}`
```

Concatenating Strings

```
let firstName ="Gerry"
let lastName = "Liu"
let fullName= firstName + " " + lastName
Fullname= firstName.concat(" " ). concat(lastName)

let x =5
let myString = "x is " +x;  // "x is 5"
```

Boolean falsy values

- ""
- 0
- NaN
- 0n
- null
- undefined
- false

The "object" Type

- Instances of a class are also called objects
  typeof {name:"Gerry"};   // "object"

  let person = {
    name: "Ray",
    age: 18
  }
  key : name, age
  values: "Ray", 18
  person.name   or person["name"] to access value.

References vs. Copying

- JavaScript objects are assigned by reference

```
let myObject = { a:1, b:2};
let myOtherObject = myObject
 MyOtherObject.a=3;   // myObject.a   =>3
```

Build-in Object Functions

Object.keys() Object.values() Object.entries()

```
let myObj={
  a:1,
  b:2,
  c:3,
}
Object.keys(myObj); // ["a","b","c"]
Object.values(myObj);  // [1, 2,3]
Object.entries(myObj); // [["a",1], ["b",2],["c",3]]
```

Object.assign

let obj1={a:1, b:2}
let obj2= {c:1,d:4}
Object.assign(obj1,obj2);
obj1 => {a:1,b:2,c:3,d:4}

can pass multiple objects to assign
Object.assign(obj1,obj2, obj3, obj4....);
Properties of objects that come later will overwrite the properties of objects that come before with same key names
Create a deep copy object
let obj3 = Object.assign({}, obj1)

ES6+ Object Destructuring

```
let person = {
   name: "Ray",
      age:  18
  }

let {name,age}= person

With default value
let {name,age, eyeColor="unknown"}= person

destructuring array
let numbers=[1,2,3,4,5]
let [x,y,z]= numbers    // x=1, y=2, z=3
```

JavaScript Arrays

let myArray = [1, 2, "Three", {message:"hello"}]

Arrays are "objects"

typeof [1, 2, 3] // => "object"

myArray[0] => 1

Build-in Array Functions

- Push and pop
  numbers=[1,2,3,4]
  numbers.push(5); // [1,2,3,4,5]
  let last=numbers.pop(); // last is 5
- Splice
  array.splice(startIndex, removeHowMany, ... elementsToAdd)
  numbers.splice(2,1) ; // [1,2,4]
  numbers.splice(2,0,100); // [1,2,100,4]
  numbers.splice(0,2, 'one', 'two'); // ["one", "two", 100,4]
- IndexOf
  numbers.indexof(3); // return 2

Build-in Array Functions

- Find

```
numbers=[1,2,3,4,5,6]
numbers.find(function(x) {return x>3})  // 4
```

- Filter

```
evenNumbers= numbers.filter(function(x) {return x%2===0})
// evenNumbers = [2,4,6]
```

- Map

```
doubleNumbers = numbers.map(function(x){ return x*2})
```

array.reduce(…..)

array.**sort**()

array.some()

array.every()

The "function" type

```
function add(x,y){ return x+1}
  typeof add;  // => "function"


let add = function(x,y) { return x+y}
```

ES6 arrow function

```
let add = (x,y) => { return x+y)
```

Arrow function can only be defined using let,var,or const.

Arrow Functions

- ES6 arrow function
  Arrow function can only be defined using let,var,or const.

```
let add = (x,y) => { return x+y}
```

With only one argument:

```
let myFunction=arg1=>{
}
```

With only one statement in the body

```
let double=x=> x*2
```

Arrow Functions to return an object

```
let someFunction=() =>{
    message: "hello",
    Time: "8:00am",
}    // this will throw error

let someFunction=() =>({
     message: "hello",
     Time: "8:00am",
})     // this is good and return an object.
```

Don't use Arrow Functions as object value

```
let myObj = {
  name:"Bob",
  logName: ()=>{
    console.log(this.name);
  }
}
myObj.logName();   // undefined
let myObj = {
   name:"Bob",
   logName: function(){
     console.log(this.name);
   }
}
myObj.logName();   // works
```

Arrow Functions can be used in callback

fetchData(data=>...)

arr.map(x=>x*2);
arr.filter(x=>x.isCompleted);

Arrow Function Default Arguments

```
Let myFunc= (x="default!", y= 100) =>{

}
let defaultArgs = (arg1="Hello",arg2=3, arg3=true) =>({
  arg1,
  arg2,
  arc3,
})
defaultArgs()
defaultArgs("GoodBye")
```

ES6+ The Spread Operator

```
let obj1={a:1, b:2}
let obj2= {c:3,d:4}
let combined = {...obj1,...obj2, e:5,f:6} //   {a:1,b:2,c:3,d:4,e:5,f6}
```

Can also used in Array

```
let arr1=[1,2]
let arr2=[3,4]
let combined = [...arr1, ...arr2]
```

Pass array of elements as function arguments.

```
let func = (arg1,arg2,...rest) =>{ console.log(rest)}
func(1,2,3,4);   // prints [3,4]
```

```
let add = (x,y,z) => x+y+z
let numbers=[1,2,3]
add(...numbers);  // 6
```

The "undefined" Type

```
let x

typeof x ;   // "undefined"
typeof y ;   // "undefined"
```

JavaScript Control Flow

Equality in JavaScript

Different Data Type => Not equal
1 === 1   // true
1 === "1" // false
10 === 10n // false

"Double equals" does not check for type
1== "1" // true
10 == 10n // true

When in doubt, use the triple equals
1 === Number("1")

Object Equality

let myObj1 = {a:1}

let myObj2 = {a:1}

myObj1 === myObj2 // false

let myObj3 = myObj1;

myObj1=== myObj3 // true

DeepEqual: we need to
include lodash library _.isEqual(obj1,obj2)
or deep-equal library   deepEqual(obj1,obj2)
Or use JSON.stringify() to convert 2 objects to strings and compare the strings.

If Statements in JavaScript

```
If(someCondition){
    // do something
} else if(otherCondtion){
   // do something else
}  else {
   // do another thing
}
```

For Loops in JavaScript

```
for( let i= 0;i<arr.length; i = i+1){
   console.log(arr[I])
}
for (let item of arr) {
   console.log(item)
}
let person = {
  name:"Ray",
  Age:18,
}
for (let key in person){
  console.log( key + ":" +  person[key])
}
arr.forEach(function(x){
   console.log(x)
});
```

While loop

```
while(someCondition){
 //do something
}
do{
//do something
} while(someCondtion)
```

The Try-Catch Block

```
try{
  //code that might fail
} catch(err){
  // error handling logic
}
```

The Switch-Case Statement

```
switch(userAnswer){
   case "a":
        // do something
        break;
   case "b":
        // do something
          break;
   default:
        // do something
}
```

The Ternary Operator

```
let greeting = isBeforeNoon? "Good Morning":"Good Afternoon";
```

JavaScript Classes

Creating Class Instances

```
class Person{
   consturctor (name,age){
       this.name=name
        this.age=age
   }
   someMethod(arg1, arg2){
   }
}
let person= new Person("Ray", 18)
person.name  // => "Ray"
```

There are currently no private class variables.

Subclasses & Inheritance

```
class Employee extends Person{
    constructor (name,age, salary, jobTitle){
        super(name,age)
        this.salary=salary
        This.jobTitle=jobTitle
    }
    someMethod(arg1,arg2){
    }
}
```

Subclass can override the parent class methods.

Work with Asynchronous Code in JavaScript

Callbacks in JavaScript

- JavaScript is single-threaded, but there are some techniques that make it "feel multi-threaded"
- fs.readFile("someFile.txt", fileContents=>{

  // do something with the file contents
  console.log("inside callback")

});
  console.log("outside callback")

The rest of program will move on while our asynchronous operstions complete
prints "outside callback", THEN "inside callback"

Use callbacks to handle asynchronous operations

```
function doSomething() {
    setTimeout( () =>{
        console.log("Here!");
    }, 2000)
}

doSomething()
```

Here!

Callback Hell

```
readFile("someFile.txt", content => {
   postRequest('www.someapi.com', content, response =>{
         //...
         updateDataOnServer(data=>{
            //...
            getUpdateData(data=>{
              //...
            })
         })
   })
})
```

With callback, any results that you get from an asynchronous operation are only accessible inside the callback, this makes things a bit difficult when writing software where there are many asynchronous operations that take place one after the other. And all of this leads to what's become known as callback hell that is a series of nested callbacks that slowly move over to the right the deeper you get and become unreadable very quickly.

Promises in JavaScript

- A nicer way to write back-to-back asynchronous operations
```
readFile('someFile.txt')
.then(contents=>{
    // send data to server
}).then(response=>{
    // update on server
}).then (data=>{
    // get update server data
}).then(data=>{
    //whatever else you need to do
})
```

The 3 possible states of Promises

- Pending – the asynchronous operation still hasn't been completed yet
- Fulfilled- the operation completed successfully
- Rejected- the operation failed.

Pending to Fulfilled
or to Rejected.

Learn about promises

```
let myPromise = new Promise ((resolve, reject) =>{
    SetTimeout(()=>{
        resolve("Success")
        // reject("error")
    }, 1000)
})
myPromise.then(message=>{
   console.log(message)
}).catch(err=>{
   console.log(err)
}).finally(()=> {
    console.log("I'm done")
    // this executes no matter what, we can do clean up logic.
})
after one second, we can see "Success!"
```

Async/Await in JavaScript

- The goal is to write asynchronous code that looks synchronous

Instead of:

```
fs.readFile("someFile.txt", contents=>{
  //...
})
readFile.("someFile.txt")
.then(contents => {
   //..
});
```

We can do

```
var content = await readFile("someFile.txt")
```

Async/Await in JavaScript

```
//promise
fetch('www.someapi.com/data')
.then(response=>{
        return response.json()
}).then(data=>{
   console.log(data)
});

//async await
let response = await fetch('www.someapi.com/data')
let data= await response.json()
console.log(data)
```

Async/Await in JavaScript

```
try{
   let response = await fetch('www.someapi.com/data')
   let data= await response.json()
   console.log(data)
} catch(error){
    console.log("help!, error occurred!")
} finally{
    // do some cleanup.
}
```

Async/Await in JavaScript

"await" can't be used directly in a JavaScript file

It must be inside a function for example
let x= await someAsyncFunction()

Any function that contains the "await" keyword must be labelled with the "async" keyword

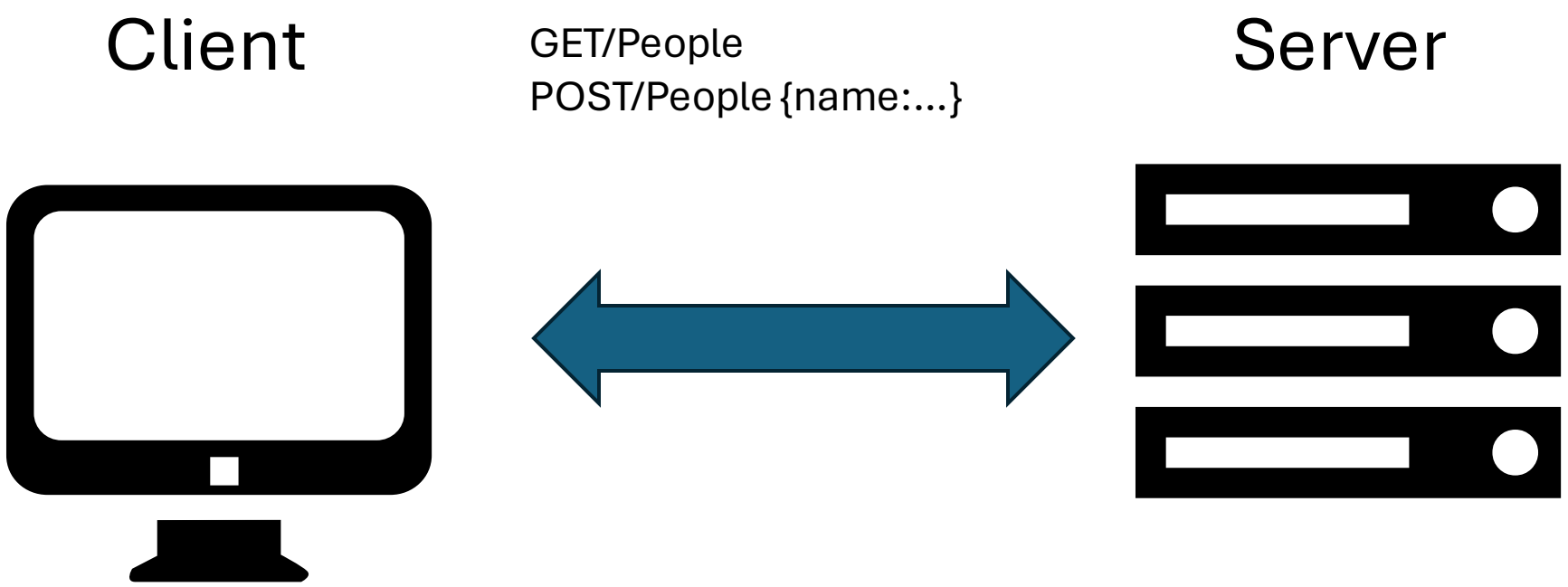
```
let doSomeAsyncStuff = async function() {
    let x = await someAsyncFunction()
}
let asyncFunction = async() => { ... }
```

Async/Await sample

```
function doSomething(){
   return new Promise ((resolve, reject) =>{
       SetTimeout(()=>{
            resolve("Success")
            // reject("error")
     }, 1000)
  })
}
async function execute() {
  try {
       let message = await doSomething();
       console.log(message)
  } catch(err ) {
     console.log(err)
  } finally{
      console.log("I'm done")
}
```

Create a Web Server with JavaScript

REST API



A REST API is a standardized interface for the client to manage data to the server

Our Server

Our Server

GET /hello	Sends back "Hello!"
GET /people	Sends an array of people objects
GET /people/:name	Sends back a specific person
GET /file-data	Reads a file and sends back contents
POST /people	Adds a new person to the server

Create and set up a Node.js Project

Create node-rest-api work dir.
mkdir node-rest-api

cd node-rest-api

npm init –y

npm install express

npm install @babel/core @babel/preset-env @babel/node
git init
add new file .gitignore to exclude node_modules/
add new file .babelrc

```
{
  "presets": ["@babel/preset-env"]
}
```

Create and run a basic Express server

- Create src dir and create server.js
- In server.js to create express server

```
import express from 'express'
let app = express()
```

- Start the server

app.listen(port, callback)

- ES5 to run the server

node src/server.js

- ES6 with bable to run the Server

npx babel-node src/server.js

Add "start": "npx babel-node src/server.js" into package.json script object

Use "npm start" to start server

- Go to brower to test

http://localhost:3000/hello

Create and test a GET endpoint

- app.get('/people',(req,res)=>{
- res.json(people)
- })
- app.get('/people/:name', (req, res) =>{
- let {name} = req.params
- let person = people.find(x=>x.name === name)
- res.json(person)
- })

Read File with FS package

```
import {promises as fs } from 'fs'
.....
app.get('/file-data', async (req, res) =>{
    let data= await fs.readFile(__dirname + "/people-data.json")
    let people = JSON.parse(data)
    res.json(people)
})
```

Create a test a POST endPoint

- Modify data with POST
  POST method allows to send extra data along with the request

npm install body-parser

```
import bodyParse from 'body-parser'
```

BodyParser takes the extra data that the client sends along with their request and puts it on the request argument of the post endpoint.

Or add

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));


app.post("/people", (req,res) => {
    let newPerson = req.body
    people.push(newPerson)
    res.json(people)
})
```