Node.js Essential Training

The Global Object:

If you're use to using JavaScript in the browser, then you're probably pretty use to the window being the global object. In node.js, the global object is global. Here we're looking at the node.js api.
https://nodejs.org/api/globals.html

//global.js
console.log("Hello world");//Hello world
The console object is available to us globally because it is a part of the global name space. So, by adding the global name space before this console log, this will actually work the same.
global.console.log("Hello world");//Hello world
var hello ="hello from node js";
console.log(global.hello);//undefined because the scope is restricted to only module

we can use javascript methods also for example I want to slice the string like
var hello ="hello world from node js";
console.log(hello.slice(17));//prints only node js
console.log(__dirname);//prints current directory path
console.log(__filename);//prints current file path

var path = require("path");
console.log(path.basename(__filename));//global.js

Argument variables with process.argv:

One important object that is available to us globally is the process object. It can be accessed from anywhere, and it contains functionality that allows us to interact with information about the current process instance. We can use the process object to get environment information, read environment variables, communicate with the terminal, or parent processes, through standard input and standard output. We can even exit the current process. This object essentially gives us a way to work with the current process instance. One of the the things that we can do with the process object is to collect all the information from the terminal, or command prompt, when the application starts.

All of this information will be saved in a variable called process.argv which stands for the argument variables used to start the process.

console.log(process.argv);

execute the node js file like node global.js
if you pass some mor arguments from the cmd like node globle .js --user chandra --greeting "Hello sir"
console.log(process.argv);//contains our user flag and a value for the user, and a greeting flag, and a value for the greeting.
to read external data which is send from command prompt:
---
function grab(flag){

```
    var index = process.argv.indexOf(flag);
    return (index===-1)?null:process.argv[index+1];
}
var greeting = grab("--greeting");
var user = grab("--user");
if(!user || !greeting){
    console.log("No external data");
}
else{
    console.log("Welcome to ",user,greeting);//Welcome to  chandra Good Morning
}
```

run with below command
node .\index.js --user chandra --greeting "Good Morning"

Standard Input and Standard Output:

Another feature of the process object is standard input and standard output. These two objects offer us a way to communicate with a process while it is running.
For now, we will use these objects to read and write data to the Terminal. Later on in the course, we're going to use the standard input and standard output objects to communicate with a child process.

```
process.stdout.write("Hello");
process.stdout.write("\n\n\n\n");


//index.js
var questions = [
    "What is your name ?",
    "What is your hobby ?",
    "What is your favorate sport ?"
];
function ask(i){
    process.stdout.write(questions[i]);
    process.stdout.write("\n\n\n");

}
ask(0);

//index.js(stdin-->writing to terminal
var questions = [
    "What is your name ?",
    "What is your hobby ?",
    "What is your favorate sport ?"
];
function ask(i){
    process.stdout.write(questions[i]);
    process.stdout.write("\n\n\n");

}
process.stdin.on("data",function(data){
```

```
    process.stdout.write("\n\n"+data.toString().trim()+"\n\n")
})
ask(0);


//save answers array
var questions = [
    "What is your name ?",
    "What is your hobby ?",
    "What is your favorate sport ?"
];
var answers = [];
function ask(i){
    process.stdout.write(questions[i]);
    process.stdout.write("\n\n\n");

}
process.stdin.on("data",function(data){
    answers.push(data.toString().trim());
    if(answers.length<questions.length){
        ask(answers.length);
    }
    else{
        process.exit();
    }
});
process.on("exit",function(){
    process.stdout.write("\n\n\n\n");
    process.stdout.write(answers[0] + " hobby is "+answers[1]+" and sport is "+answers[2]);
    process.stdout.write("\n\n\n\n");
})
ask(0);
```

Timing functions in node js:

In the last lesson we started working with Node.js asynchronously by using event listeners. Another way we can work with Node.js asynchronously is through using the timing functions. The timing functions setTimeout, clearTimeout, setInterval, and clearInterval work the same way they do in the browser and are available to you globally.

setTimeout:

---

setTimeout will create a delay of a certain time and then invoke a callback function.

```
const eventEmitter = require("events");
const e = new eventEmitter();

console.log("start");
```

```
process.nextTick(function(){
    console.log("nextTick:after current process");
});
setTimeout(function(){
    console.log("SetTimeout:End of event loop");
},0);
setImmediate(function(){
    console.log("SetImmediate:start of next event loop");
})

e.on('event-1',()=>{
    console.log("1");
});
e.on("event-2",()=>{
    console.log("2");
});
e.on("event-3",()=>{
    console.log(3);
});

e.emit('event-1');
e.emit('event-2');
e.emit('event-3');

console.log("end");
//output
start
1
2
3
end
nextTick:after current process
SetTimeout:End of event loop
SetImmediate:start of next event loop
```

```
//
var waitTime = 7000;
console.log("started");

var intr = setInterval(function(){
   var current = new Date().toLocaleTimeString();
   console.log(current);
},1000);


setTimeout(function(){
   clearInterval(intr);
   console.log("end of event loop");
},waitTime);


core modules:
-------------
var path = require("path");
console.log(path.basename(__filename));//index.js
var dirUploads = path.join(__dirname,"www","files","uploads");
console.log(dirUploads);//D:\devops_angular6\chat\www\files\uploads

var util = require("util");
util.log(path.basename(__filename));//18 Oct 14:40:20 - index.js

var v8 = require("v8");
util.log(v8.getHeapStatistics());//will print memory details
```

creating child process with exec:

Node.js comes with a Child Process module which allows you to execute external processes in your environment. In other words, your Node.js app can run and communicate with other applications on the computer that it is hosting.

exec:

```
var exec = require('child_process').exec;
exec("dir",function(err,stdout){
   if(err){
      console.log(err);
   }
   else{
      console.log('listing')
      console.log(stdout);
   }
});
//exec
var exec = require('child_process').exec;
exec("git --version",function(err,stdout){
   if(err){
      console.log(err);
   }
   else{
      console.log('git version')
      console.log(stdout);
   }
```

```
});
```

The File System:

Node.js also ships with a module that allows us to interact with the file system. The fs module can be used to list files and directories, create files and directories, stream files, write files, read files, modify file permissions or just about anything that you need to be able to do with the file system.

Listing Directory files:

```
var fs = require("fs");
var files = fs.readdirSync('./lib');//read directory files synchronously,here block will come because all other operation should hold untill this completes
console.log(files);
```

Asynchronous:

```
var fs = require("fs");
fs.readdir('./lib',function(err,files){
  if(err){
    console.log(err);
  }
  else
  {
    console.log(files);
  }
});
console.log("reading files....!")
```

Read files:

```
var fs = require("fs");
var contents = fs.readFileSync("./lib/ex.txt","UTF-8");
console.log(contents);
```

Asynchronous:

```
var fs = require("fs");
fs.readFile("./lib/ex.txt","UTF-8",function(err,contents){
   console.log(contents);
});


//
var fs = require("fs");
fs.readFile("./lib/ex.txt",function(err,contents){
   console.log(contents);//buffer format will read
});

//
var fs = require("fs");
var path = require("path");
fs.readdir("./lib", function (err, files) {
   if (err) {
      console.log(err)
   }
   else {
      files.forEach(function (fileName) {
         var file = path.join(__dirname,"lib",fileName);
```

```
        var stats = fs.statSync(file);//find statisstics of the file
        if(stats.isFile() && fileName==".Ds_store"){
           fs.readFile(file,"UTF-8",function(err,contents){
              console.log(contents);


           })
        }
      })
   }
})
```

Writing and appending files:

Another feature of the file system module is the ability to create new files, to write text or binary content to those files, or to append text or binary content to an existing file.

```
var fs = require("fs");
var md = "hello new file"
fs.writeFile("sample.txt",md.trim(),function(err){
   console.log("file created");
});
```

append:contents appends to existing file

```
var fs = require("fs");
var md = "chandra pothireddy"
fs.appendFile("sample.txt",md.trim(),function(err){
```

```
    console.log("file contents added");
});
```

Directory creation:

```
var fs = require("fs");
fs.mkdir("./newlib",function(err){
    console.log("new lib created");
});
//checks for directory there or not

var fs = require("fs");

if (fs.existsSync("./newlib")) {
    console.log("directory already there");
}
else {
    fs.mkdir("./newlib", function (err) {
        console.log("new lib created");
    });
}
```

Renaming and removing files:

```
var fs = require("fs");
fs.renameSync("./lib/index.html","sample.html");
console.log("file renamed");

fs.rename("sample.txt","chandra.txt",function(err){
```

```
  if(err){
     console.log("error in rename file");
  }
})

remove:
-------
var fs = require("fs");

try{
  fs.unlinkSync("./lib/.Ds_store");
}
catch(err){
  console.log(err);
}
fs.unlink("sample.html",function(err){
  if(err){
     console.log(err);
  }
  else{
     console.log("sample removed")
  }
})

Rename and remove directories:
------------------------------
var fs = require("fs");
fs.renameSync("./lib","./vendor");
console.log("directory renamed");
```

remove directory:

---

if you want to remove directory, first remove all files inside that directory....if the directory is empty. we can remove directly.

```
var fs = require("fs");
fs.rmdir("./newlib",function(err){
  if(err){
    throw err;
  }
  else{
    console.log("Directory removed succes");
  }
});
//removed directory which is contains inside some files
var fs = require("fs");
fs.readdirSync("./vendor").forEach(function(fileName){
  fs.unlinkSync("./vendor/"+fileName);
})

fs.rmdir("./vendor",function(err){
  if(err){
    throw err;
  }
  else{
    console.log("Vendor directory removed");
  }
});
```

Readable file streams:

Streams give us a way to asynchronously handle continuous data flows. Understanding how streams work will dramatically improve the way your application handles large data.

-> Streams in Node.js are implementations of the underlying abstract extreme interface

If we read large file via file system:

```
var fs =require("fs");
fs.readFile("index.js","UTF-8",function(err,contents){
  if(contents){
    console.log(contents.length);
  }
})
```

the problem is readFile waits until the entire file is read before invoking the call back and passing the file contents.

It also buffers the entire file in one variable. If our big data app experiences heavy traffic, read file is going to create latency and could impact our memory. So a better solution might be to implement a readable stream.

```
var fs =require("fs");
var stream = fs.createReadStream("index.js","UTF-8");

stream.once("data",function(){
  console.log("\n\n\n");
  console.log("Reading started");
  console.log("\n\n\n");
});

var data = '';
```

```
stream.on("data",function(chunk){
   process.stdout.write(`chunk:${chunk.length} |`);//Great, so now, as opposed to waiting for the entire file to be read, we can use this stream to start receiving small chunks of data from this file.
   data +=chunk;
});

stream.on("end",function(){
   console.log("\n\n\n");
   console.log(`finshed data length is ${data.length}`);
   console.log("\n\n\n");
})
```

So we've wired up three listeners to our stream. A listener that will listen one time for a data event and that will let our users know that we have started reading the file, a listener that will listen for every data event that is raised and it will gather all of the text chunks of data and concatenate our data variable, and then a listener that will listen for the end event on the stream and it will show us the length of the variable that we have concatenated.

So this is a very different way to go about reading the file but it means that we do not have to wait to buffer all of the data at once, that we can start receiving the text in this file chunk by chunk by chunk and then we can put together those data chunks to eventually have our full file.

writable file streams:

The writable stream is used to write the data chunks that are going to be read by the readable streams

```
var fs = require("fs");
var stream = fs.createWriteStream("chandra.txt");
stream.write("chandra streams implementation");
```

The HTTP Module:

---

we're going to dive deep into Node.js's HTTP module. The HTTP module will be used for creating web servers, for making requests, for handling responses. There are two modules for this. There's the HTTP module and the HTTPS module. Now, both of these modules are very, very similar, but we'll only use the HTTPS module when we're working with a secure server. So that means if we want to create an HTTPS server, we would use the HTTPS module, and then we would have to supply the security certificate.

With the HTTP module, there's no need to supply a security certificate.

Making a request:

```
var https = require("https");
var options = {
   host:"en.wikipedia.org",
   port:443,//for http public sites port is 80
   path:"/wiki/Sachin_Tendulkar",
   method:"GET"
};
var req = https.request(options,function(res){
   console.log(`response status code: ${res.statusCode}`);
});
req.on("error",function(err){
   console.log(`problem with request : ${err.message}`)
})
req.end();
```

Build a web server:

One of the coolest things that we can do with Node.js is build web servers. Node.js is JavaScript, the language of the web. So of course, one of the most important features is the ability to create web servers or you can additionally use the HTTPS module that ships with Node.js. Both of these modules will allow you to create web servers, but if you need to create a secure web server, you will need to add a security certificate to the HTTPS module.

```
var http = require("http");
var server = http.createServer(function(req,res){
   res.writeHead(200,{"Content-Type":"text/plain"});
   res.end("Hello world");
});
server.listen(3000);
console.log("Http server listening on port 3000");
```

One of the coolest things that we can do with Node.js is build web servers. Node.js is JavaScript, the language of the web. So of course, one of the most important features is the ability to create web servers or you can additionally use the HTTPS module that ships with Node.js. Both of these modules will allow you to create web servers, but if you need to create a secure web server, you will need to add a security certificate to the HTTPS module.

if we want chage response type is html then

```
var http = require("http");
var server = http.createServer(function(req,res){
   res.writeHead(200,{"Content-Type":"text/html"});
   res.end(`
   <h1>chandra work</h1>
   `);
});
server.listen(3000);
console.log("Http server listening on port 3000");
```

Serving files:

One of the most important features of web servers is the ability to serve files. Usually, the first thing we learn about Apache or IIS is where to put our HTML, CSS, images and other assets that we want that web server to serve. Now, with Node.js, we actually have to build the web server. In order to build a web server, we're going to have to use the https module in conjunction with the fs module. When we request a file, we are going to have to use the file system to load that file, and then we can use the HTTP module to respond with its contents.

```
var http = require("http");
var fs = require("fs");
var path = require("path");

http.createServer(function(req,res){
   console.log(`${req.method} requested ${req.url}`);
   if(req.url ==="/"){
      fs.readFile("index.html","UTF-8",function(err,html){
         res.writeHead(200,{"Content-Type":"text/html"});
         res.end(html);
      });
   }
   else{
      res.writeHead(404,{"Content-Type":"text/plain"});
      res.end("File not found");
   }
}).listen(3000);
console.log("server is running on port 3000");
```

Serving JSON data:

We can also use the HTTP module to create an HTTP API, or a server whose primary purpose is to serve JSON data. APIs are used to serve data to client applications. These applications typically include mobile apps, and single page websites, but, any client who can make an HTTP request can communicate with an API.

```
//data.json
[
  {
    "name":"chandra",
    "address":"AP"
  },
  {
    "name":"babu",
    "address":"HYD"
  }
]
//server.js
var http = require("http");
var data = require("./data.json");
http.createServer(function (req, res) {
  res.writeHead(200, { "Content-Type": "text/json" });
  res.end(JSON.stringify(data));
}).listen(3000);
console.log("listining on 3000 port");
```

Collecting POST data:

So far we have created servers using the http module that only handled GET requests. We can use the http module to create servers that also handle POST requests, PUT requests, DELETE requests, and many others

```
//server.js
var http = require("http");
var path = require("path");
var fs = require("fs");
http.createServer(function(req,res){
  if(req.method =="GET"){
    fs.readFile('form.html',"UTF-8",function(err,html){
      res.writeHead(200,{"Content-Type":"text/html"});
      res.end(html);
    })
  }
  if(req.method=="POST"){
    console.log("in post req");
    var body = '';
    req.on("data",function(chunk){
      body += chunk;
    });
    req.on("end",function(){
      res.writeHead(200,{"Content-Type":"text/html"});
      res.end(`
      <p> response is${body}</p>
      `);
    })
  }
```

```
}).listen(3000);
console.log("http server listing on port 3000");

//form.html
<form action="/" method="post">
<input type="text" name="first" />
<butt>send</butt>
</form>
```

Web servers:

Express is a very popular framework for developing web server applications.

create package.json:

---

npm init will start to build this package.json file

npm install pkg-name --save

npm remove pkg-name --save

package.json symbols:

In the simplest terms, the tilde matches the most recent minor version (the middle number). ~1.2.3 will match all 1.2.x versions but will miss 1.3.0.

The caret, on the other hand, is more relaxed. It will update you to the most recent major version (the first number). ^1.2.3 will match any 1.x.x release including 1.3.0, but will hold off on 2.0.0.

Introduction to express framework:

```
var express = require("express");
var app = express();
app.use(express.static('./public'));
app.listen(3000);
console.log("express server is running on port 3000");
module.exports = app;
```

```
//create some custom middleware
var express = require("express");
var app = express();
//add some custom middleware here
app.use(function(req,res,next){
   //this middleware execute before all routing calls
   console.log(`${req.method} requested ${req.url}`);
   next();//next middleware will continue..here static file
});

app.use(express.static('./public'));
app.listen(3000);
console.log("express server is running on port 3000");
module.exports = app;
```

if no route found:

```
var express = require("express");
var app = express();
//add some custom middleware here
app.use(function(req,res,next){
   //this middleware execute before all routing calls
   console.log(`${req.method} requested ${req.url}`);

    next();//next middleware will continue..here static file
});

app.use(express.static('./public'));
```

```
//if no one route found then below one will execute
app.use(function(req,res,next){
  //this middleware execute before all routing calls
  console.log(`${req.method} requested ${req.url}`);
  res.status(404).send('This page does not exist!');
  // next();//next middleware will continue..here static file
});

app.use(function(error,req,res,next){
  console.log(error);
  res.status(500).send("Internal server error");
})
app.listen(3000);
console.log("express server is running on port 3000");
module.exports = app;
```

Websockets:

They allow for a true two way connection between the client and the server. Web Sockets use their own protocol to send and receive messages from a TCP server.

Until recently, WebSockets were not part of the step. We had no way push information from the server to the browser.

Pooling:

The browser had to constantly check the server API by making a gate request to see if the state of the server has changed. We call this polling

Long pooling:

When you have a long poll, if information does change on the server, we can immediately receive a response with the changed information. So you can think of long polling as just a more efficient way of

polling. Well, now we have Web Sockets available to us which means that we can connect to server and leave the connection open so that we can send and receive data.

Now, Web Sockets are not just limited to the browser. With Web Sockets, clients can connect to the server and leave a two way connection open. Through this connection, clients can send data that are easily broadcasted to every open connection. So the server is able to push data chain changes to the client using Web Sockets. Web Sockets are not just limited to the browser. Any client can connect to your server including native applications. Now instead of HTTP, Web Sockets use their own protocol. Settinng up a Web Sockets from scratch on your web application can be a little tricky You need a TCP Socket server and a HTTP proxy.

Fortunately, there are node marginals that will help us with building our non Web Sockets.

//index.js

```
var WebSocketServer = require("ws").Server;//get constructor function from ws module
var wss = new WebSocketServer({port:5000});//web socket is running on port 5000
wss.on("connection",function(ws){//when connection open

  ws.on("message",function(message){//when messsge triggerd
    if(message==="exit"){
      ws.close();//if client type exit need to close the connection
    }
    else{
      //broadcast to all clients
      wss.clients.forEach(function(client){
        client.send(message);//for msg for every client
      })
    }
  })
  ws.send("Welcome to cyber chat");
```

```
});

//index.html
<script src="ws-client.js"></script>
<body>
<h1>Webscockets</h1>
<div class="messages"></div>
<form action="javascript:void(0)">
<input type="text" id="message" required autofocus />
</form>
</body>


//ws-client.js
var ws = new WebSocket("ws://localhost:5000");
ws.onopen = function(){
   setTitle("Connected to cyber chat");
};
ws.onclose = function(){
   setTitle("Disconnected");
}
ws.onmessage = function(payload){
   printMessage(payload.data);
};
window.onload = function(){
   // your code
   document.forms[0].onsubmit = function(){
      var input = document.getElementById('message');
      ws.send(input.value);
```

```
    input.value="";
  }
};
function setTitle(title){
  document.querySelector('h1').innerHTML = title;
}
function printMessage(message){
  var p = document.createElement('p');
  p.innerText = message;
  document.querySelector('div.messages').appendChild(p);
}
```

If you create websockets with native JavaScript in the browser, they'll work in most modern browsers. But the support for them in older browsers isn't quite there yet.

So, one of the things that we can do, is, we can incorporate a module called Socket.IO. Socket.IO is a module that will help us build websockets, that has its own client and its own server JavaScript. What Socket.IO does, is it fails back to long polling when websockets aren't supported.

So, if websockets aren't supported in a browser, Socket.IO will still most likely work.

Working with socket.io:

Working with socket.io:

Install socket.io

npm install socket.io --save

npm install express

```
//sio-server.js
----------
var express = require("express");
var http = require("http");
var app = express();

var server = http.createServer(app).listen(3000);
var io = require("socket.io")(server);
app.use(express.static("./public"));
io.on("connection",function(socket){

   socket.on("chat",function(message){
      socket.broadcast.emit("message",message);
   })
   socket.emit("message","welcome to cyber chat");

});
console.log("socket server is connected and listening on port 3000");

//main.js--->client js file
var socket = io("http://localhost:3000")
socket.on("disconnect",function(){
   setTitle("Disconnected");
```

```
});
socket.on("connect",function(){
   setTitle("connected to cyber chat");
});
socket.on("message",function(message){
   printMessage(message);
})

window.onload = function(){
   // your code
   document.forms[0].onsubmit = function(){
      var input = document.getElementById('message');
      printMessage(input.value);
      socket.emit("chat",input.value);
      input.value="";
   }
};
function setTitle(title){
   document.querySelector('h1').innerHTML = title;
}
function printMessage(message){
   var p = document.createElement('p');
   p.innerText = message;
   document.querySelector('div.messages').appendChild(p);
}

//index.html
----------
<script src="https://cdn.socket.io/socket.io-1.0.0.js"></script>
```

```
<script src="main.js"></script>
<body>
<h1>Webscockets</h1>
<div class="messages"></div>
<form action="javascript:void(0)">
<input type="text" id="message" required autofocus />
</form>
</body>
```