CS 4032 – Web Programming

National University
Of Computer and Emerging Sciences

Collection of JavaScript based technologies used to develop web applications.

node JS™

express

What is node.js ?

- An open-source server environment
  - Uses JavaScript on the server
  - Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
  - Runs on Google's V8 JavaScript Engine
- Support **Asynchronous** Programming

Synchronous Programming

- Some tasks may take a very long time, e.g., read/write a file on disk, request data from a server, query a database …
- *Why waiting is bad??*

```
some_task(); /*
  wait for task to
  complete
*/

// process the result
// of task
… …

// do other things
… …
```

Multi-processing and Multi-threading

One process/thread

```
some_task();

// process the result
// of task
… …
```

Another process/thread

```
// do other things
… …
```

- What's the different between a process and a thread??

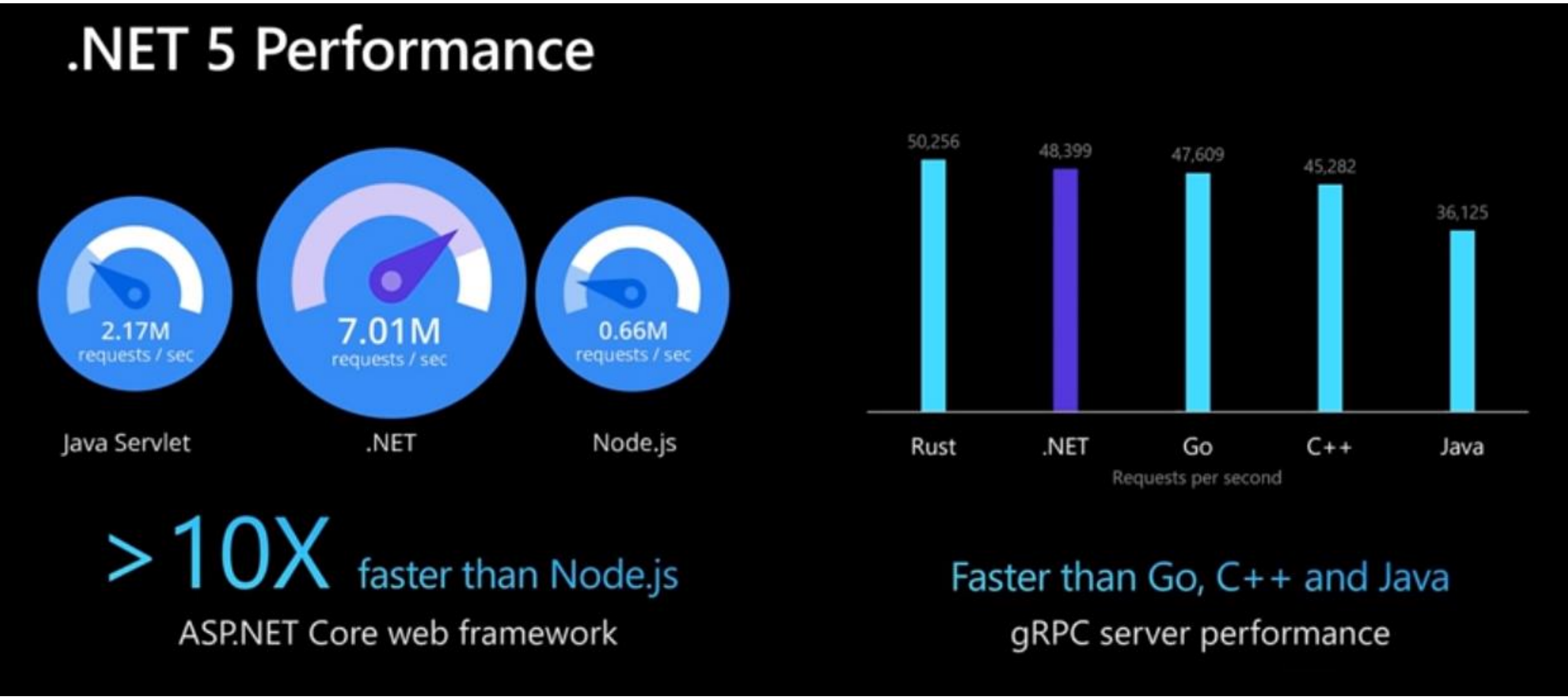
Problems of Multi-processing and Multi-threading

- OS must allocate some resources for each process/thread
- Switching between processes and threads (a.k.a. *context switch*) takes time
- Communicating among processes and synchronizing multiple threads are difficult

Big problems for busy web servers

⬇

One of the reasons why Node.js became popular in server-side development



.NET 5 Performance

>10X faster than Node.js

ASP.NET Core web framework

Faster than Go, C++ and Java

gRPC server performance

National University
Of Computer and Emerging Sciences

Asynchronous Programming

- Example : Web server
- Open a file on the server and return the content to the client

PHP, ASP, Others

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request

Node

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Asynchronous Programming …

```
callback(result) {
  // process the result
  // of task
  … …
}

some_task( callback ); /*
  calls to some_task()
  returns immediately
*/

// do other things
… …
```

… Asynchronous Programming

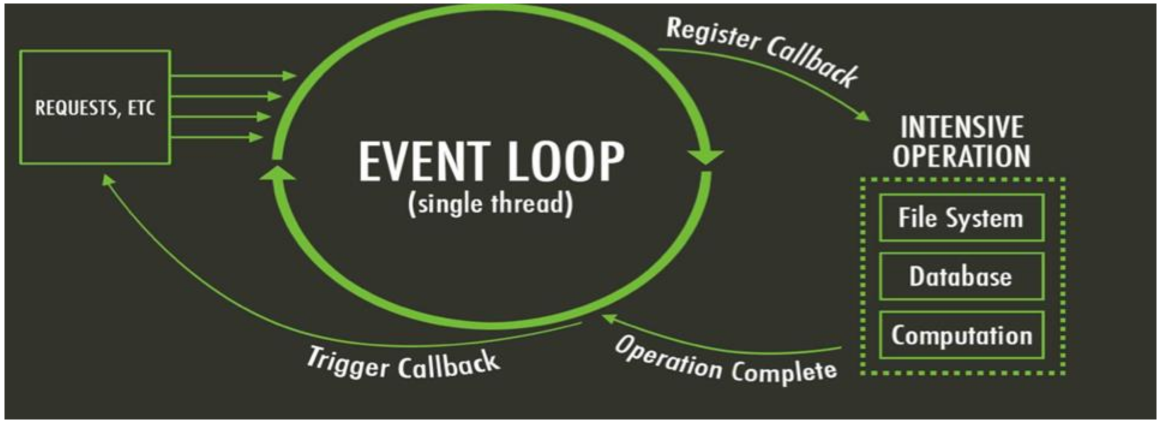
- Everything runs in one thread
- Asynchronous calls return immediately (a.k.a. *non-blocking*)
- A callback function is called when the result is ready

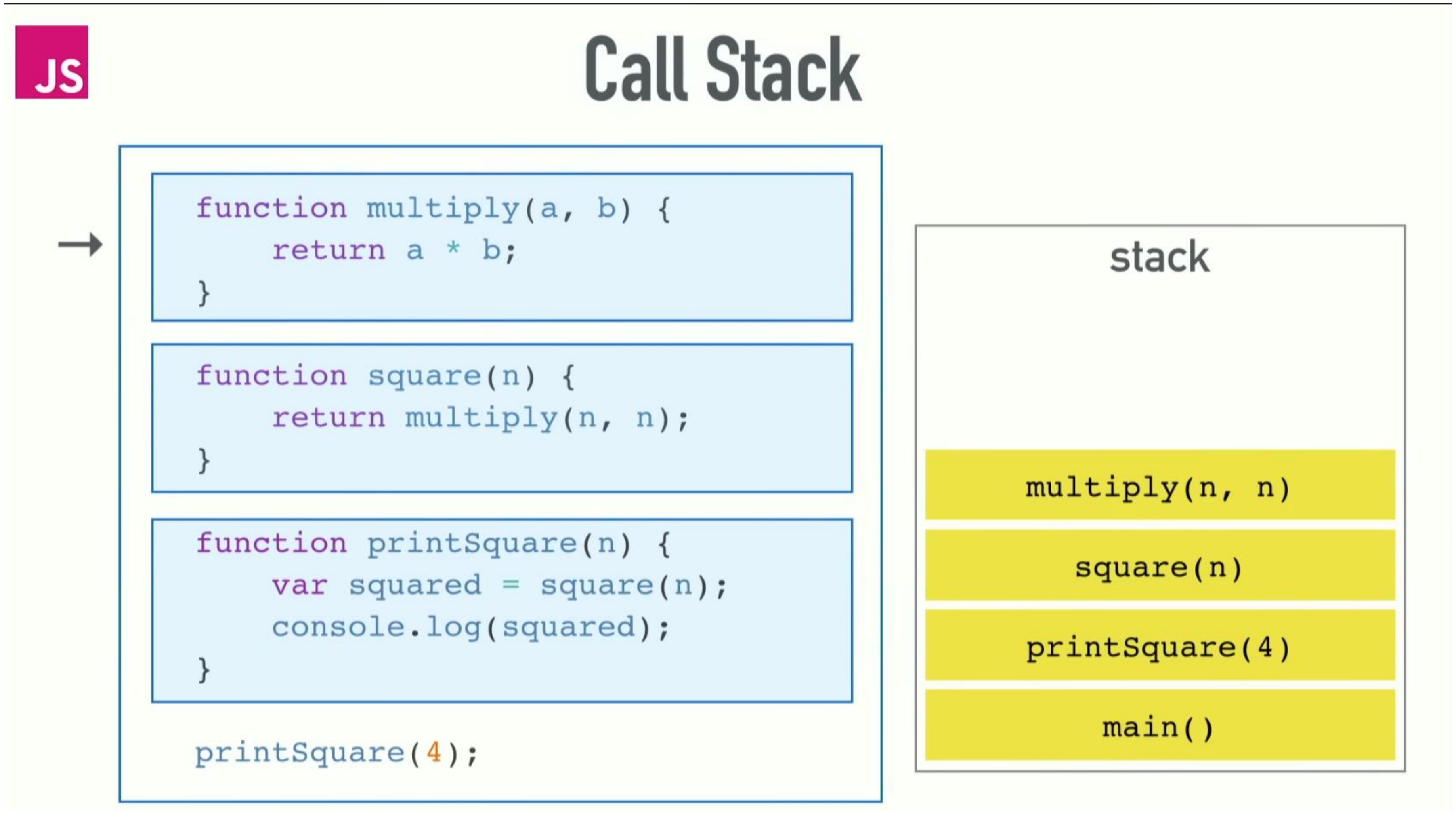
```
func(args…, callback(err,result))
```

- A.K.A. Event-driven Programming
  - A callback function is basically an event handler that handles the "result is ready" event

What is unique about Node.js?

- JavaScript on server-side thus making communication between client and server will happen in same language
- Servers normally thread based but Node.JS is “Event” based. Node.JS serves each request in a Evented loop that can handle simultaneous requests.

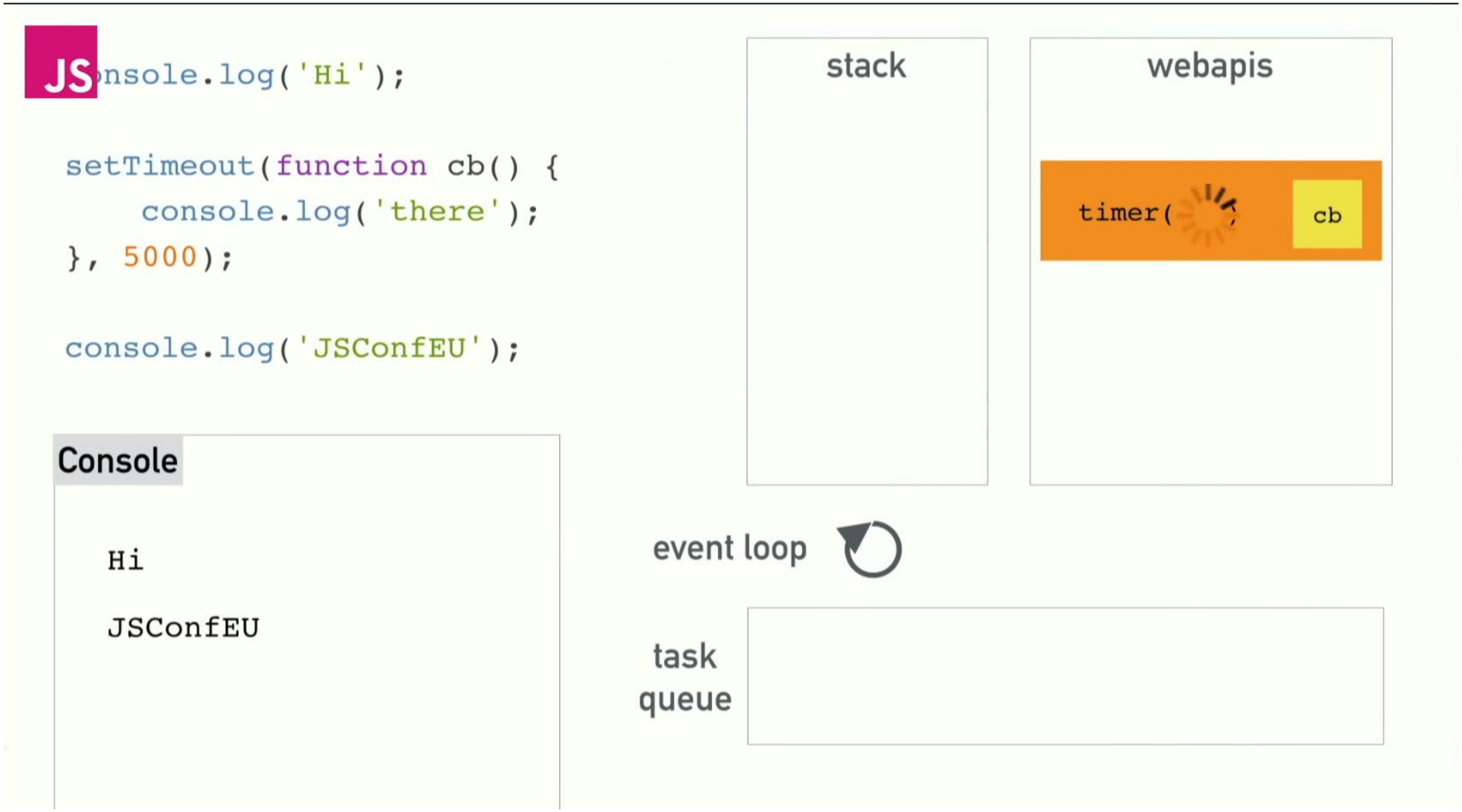




Call Stack

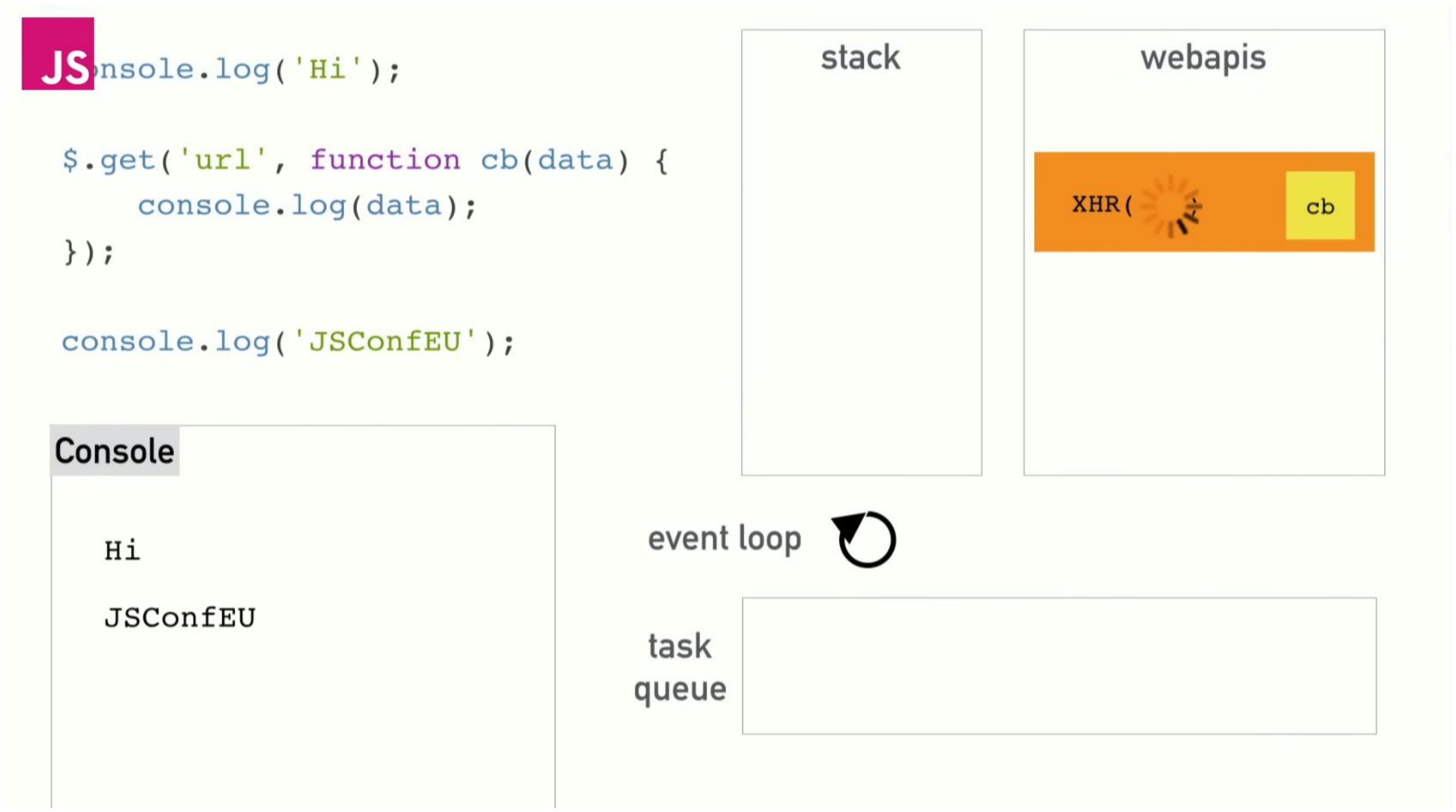
```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```

Source: https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html



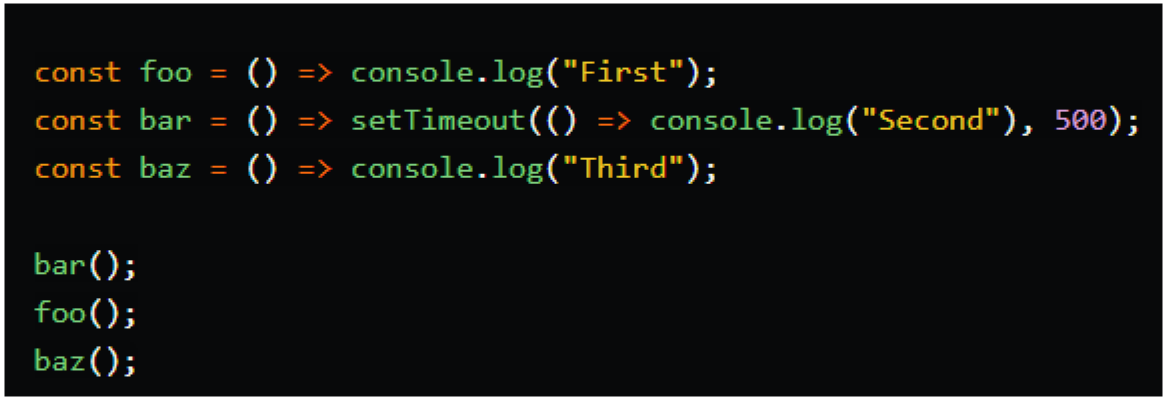
```
nsole.log('Hi');

setTimeout(function cb() {
    console.log('there');
}, 5000);

console.log('JSConfEU');
```

```
Hi

JSConfEU
```

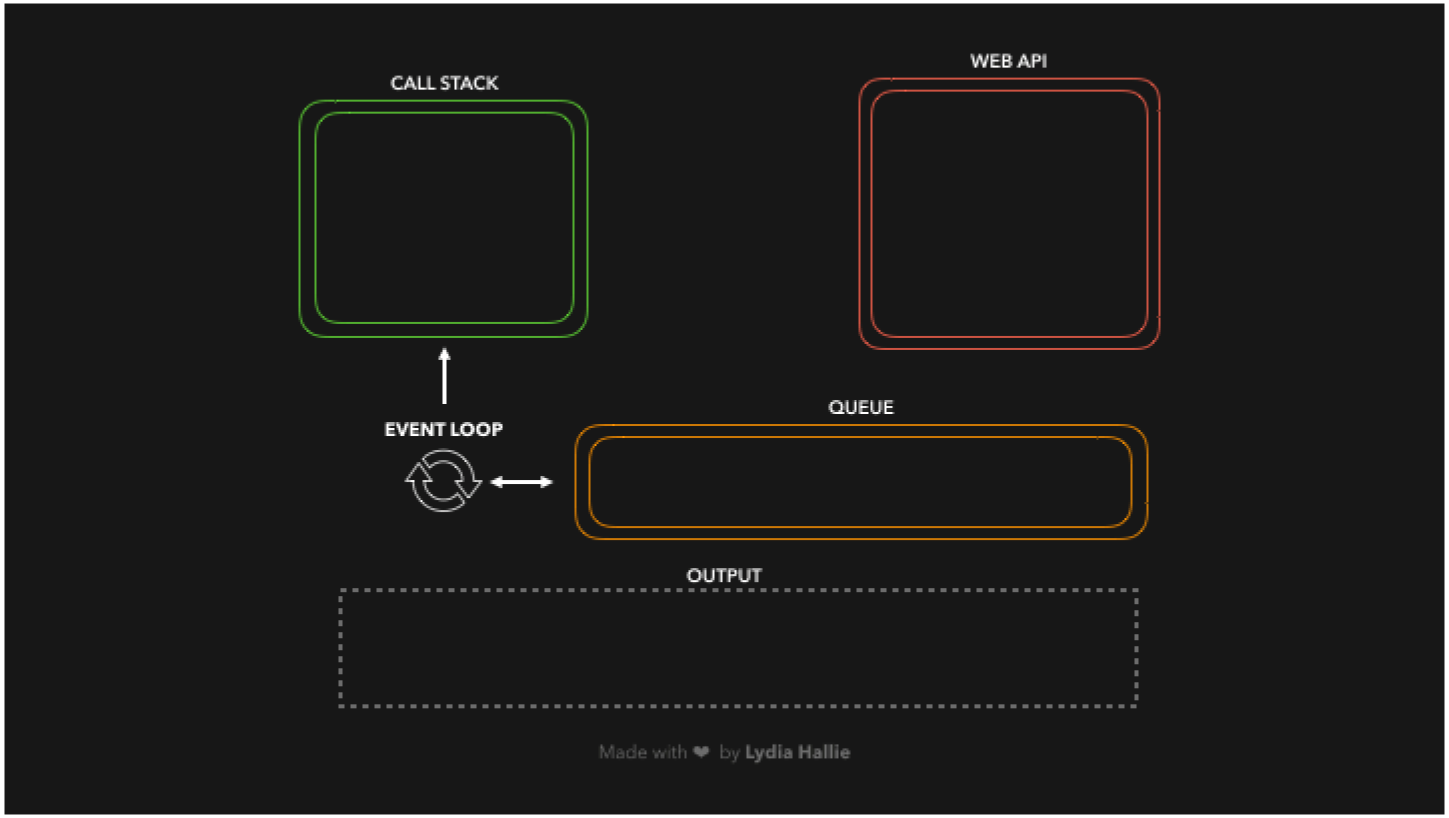


```
JSnsole.log('Hi');

$.get('url', function cb(data) {
    console.log(data);
});

console.log('JSConfEU');
```

```
Hi

JSConfEU
```



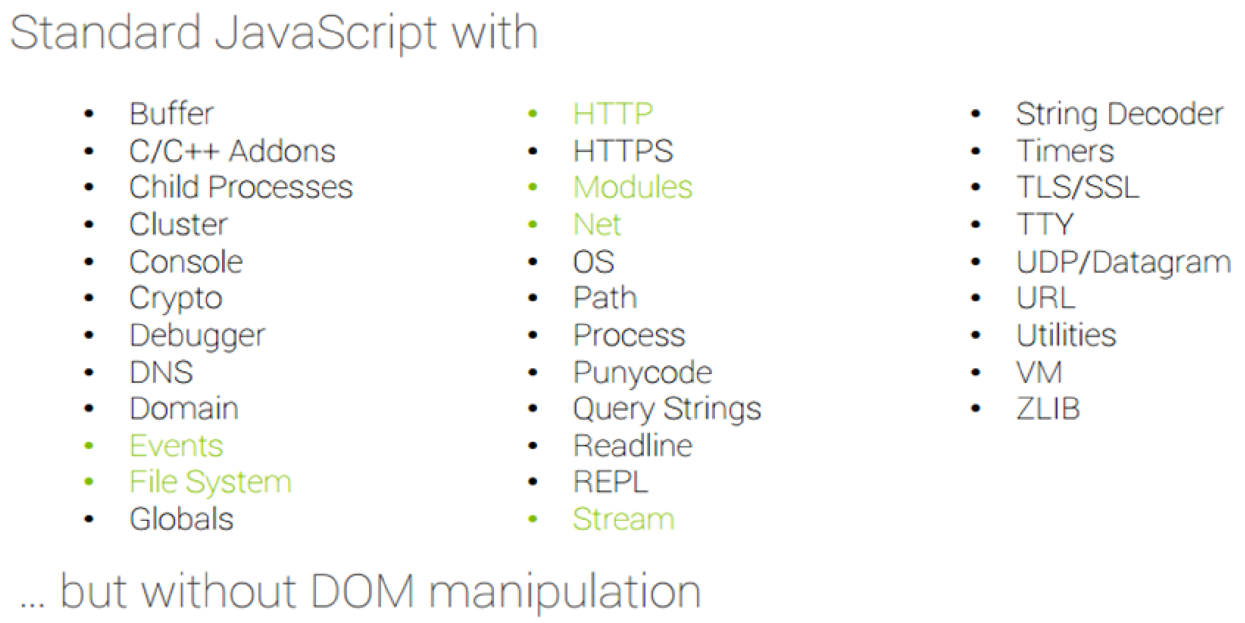
```
const foo = () => console.log("First");
const bar = () => setTimeout(() => console.log("Second"), 500);
const baz = () => console.log("Third");

bar();
foo();
baz();
```



Source: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif

Why Use Node.js ?

- Node's goal is to provide an easy way to build scalable network programs.
- It lets you layered on top of the TCP library is an HTTP and HTTPS client/server.
- The JS executed by the V8 JavaScript engine (the thing that makes Google Chrome so fast)
- Node provides a JavaScript API to access the network and file system.



Standard JavaScript with

- Buffer
- C/C++ Addons
- Child Processes
- Cluster
- Console
- Crypto
- Debugger
- DNS
- Domain
- Events
- File System
- Globals
- HTTP
- HTTPS
- Modules
- Net
- OS
- Path
- Process
- Punycode
- Query Strings
- Readline
- REPL
- Stream
- String Decoder
- Timers
- TLS/SSL
- TTY
- UDP/Datagram
- URL
- Utilities
- VM
- ZLIB

… but without DOM manipulation

What can’t do with Node?

- Node is a platform for writing JavaScript applications outside web browsers. This is not the JavaScript we are familiar with in web browsers.
- There is no DOM built into Node, nor any other browser capability.
- Node can’t run on GUI, but run on terminal
- In the Node.js module system, each file is treated as a separate module.

Installing and using node Module

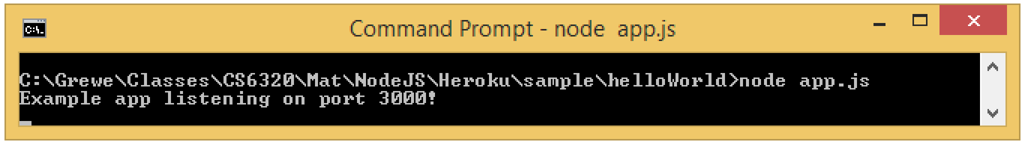
- Install a module…..inside your project directory
  - npm install <module name>
- Using module….. Inside your JavaScript code
  - var http = require(‘http’);
  - var fs = require(‘fs’);
  - var express = require(‘express’);

Hello World example (index.js)

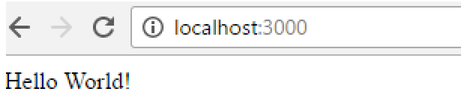
```
const http = require('http’);
const hostname = '127.0.0.1’;
const port = 3000;

const server = http.createServer((req, res) => {
       res.statusCode = 200;
       res.setHeader('Content-Type', 'text/plain’);
       res.end('Hello World’);
});



server.listen(port, hostname, () => {
       console.log(`Server running at http://${hostname}:${port}/`);
});
```

Run your hello world application

- Run the app with the following command:
  - node index.js
- Then, load http://localhost:3000/ in a browser to see the output.



```
C:\Grewe\Classes\CS6320\Mat\NodeJS\Heroku\sample\helloWorld>node app.js
Example app listening on port 3000!
```



Hello World!

Asynchronous Programming Example

- `write_file.js`: open a file, add one line, then close the file
  - Use of the File System package
    - open()
    - write()
    - close()
  - "Callback Hell" (a.k.a. "Pyramid of Doom")
- `async` declares a function to be asynchronous
  - The return value of the function will be wrapped inside a Promise
- `await` waits until a Promise settles and returns its result
  - *`await` can only be used in an `async` function*

```javascript
"use strict";

const fs = require("fs");
const util = require("util");

const fopen = util.promisify(fs.open);
const fwrite = util.promisify(fs.write);
const fclose = util.promisify(fs.close);

async function write_file() {
  try {
    let fd = await fopen("test.txt", "a");
    let result = await fwrite(fd, "A New Line!\n");
    console.log(`${result.bytesWritten} bytes written.`);
    await fclose(fd);
  } catch (err) {
    console.log(err);
  }
}

write_file();
```

Callback Hell Example

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

National University
Of Computer and Emerging Sciences

Promise

- A Promise is a JavaScript object
  - `executor`: a function that may take some time to complete. After it's finished, it sets the values of `state` and `result` based on whether the operation is successful
  - `state`: "pending" ➔ "fulfilled"/"rejected"
  - `result`: undefined ➔ value/error

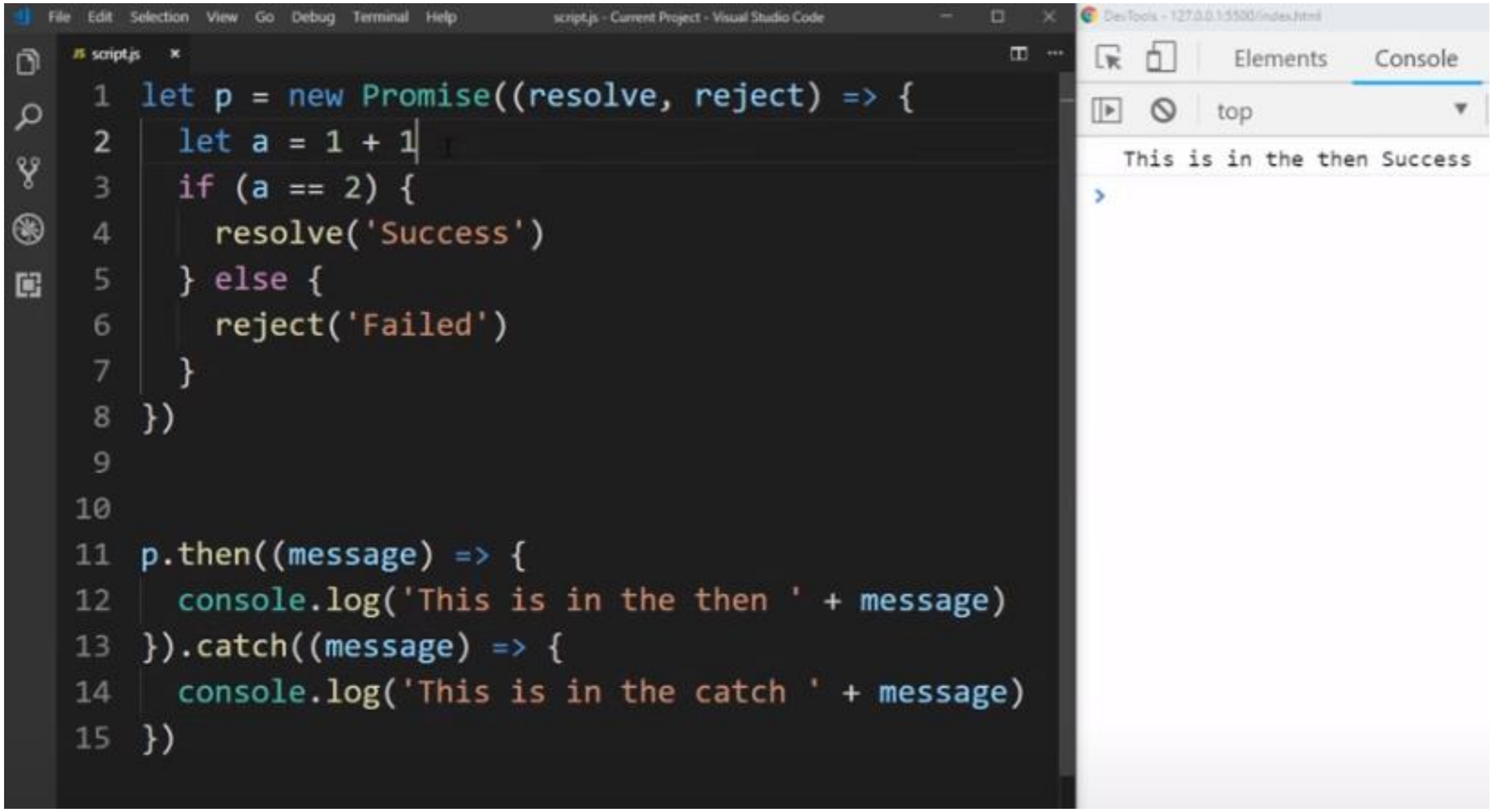
```
var promise = doSomethingAync()
promise.then(onFulfilled, onRejected)
```

Use A Promise

```
promise.then(
  function(result) {/* handle result */},
  function(err) {/* handle error */ }
);
```

- After `executor` finishes, either the success handler or the error handler will be called and `result` will be passed as the argument to the handler

Promise



File Edit Selection View Go Debug Terminal Help
script.js - Current Project - Visual Studio Code
script.js
DevTools - 127.0.0.1:5500/index.html
Elements
Console
top

```javascript
let p = new Promise((resolve, reject) => {
  let a = 1 + 1
  if (a == 2) {
    resolve('Success')
  } else {
    reject('Failed')
  }
})


p.then((message) => {
  console.log('This is in the then ' + message)
}).catch((message) => {
  console.log('This is in the catch ' + message)
})
```

```
This is in the then Success
```

National University
Of Computer and Emerging Sciences

About Promise

- There can only be one result or an error
- Once a promise is settled, the result (or error) never changes
- `then()` can be called multiple times to register multiple handlers

Other Common Usage of Promise

```
promise.then( success_handler );

promise.then( null, error_handler );

promise.catch( error_handler );

promise
   .then( success_handler )
   .catch( error_handler )
```

Promise Example

- `get_page_promise.js`: request and print a web page
  - Use of the request-promise-native package
  - `request()` returns a *promise*

```
const request2 = require("request-promise-native");

function get_page_with_promise() {
  request2("http://nu.edu.pk").then(body =>
console.log(body));
}
```

Promises Chaining

- Suppose we have three functions `f1`, `f2`, `f3`
  - `f1` returns a Promise
  - `f2` relies on the result produced by `f1`
  - `f3` relies on the result produced by `f2`

```
f1.then(f2).then(f3)
```

Understand Promise Chaining …

```
f1.then(f2)
```

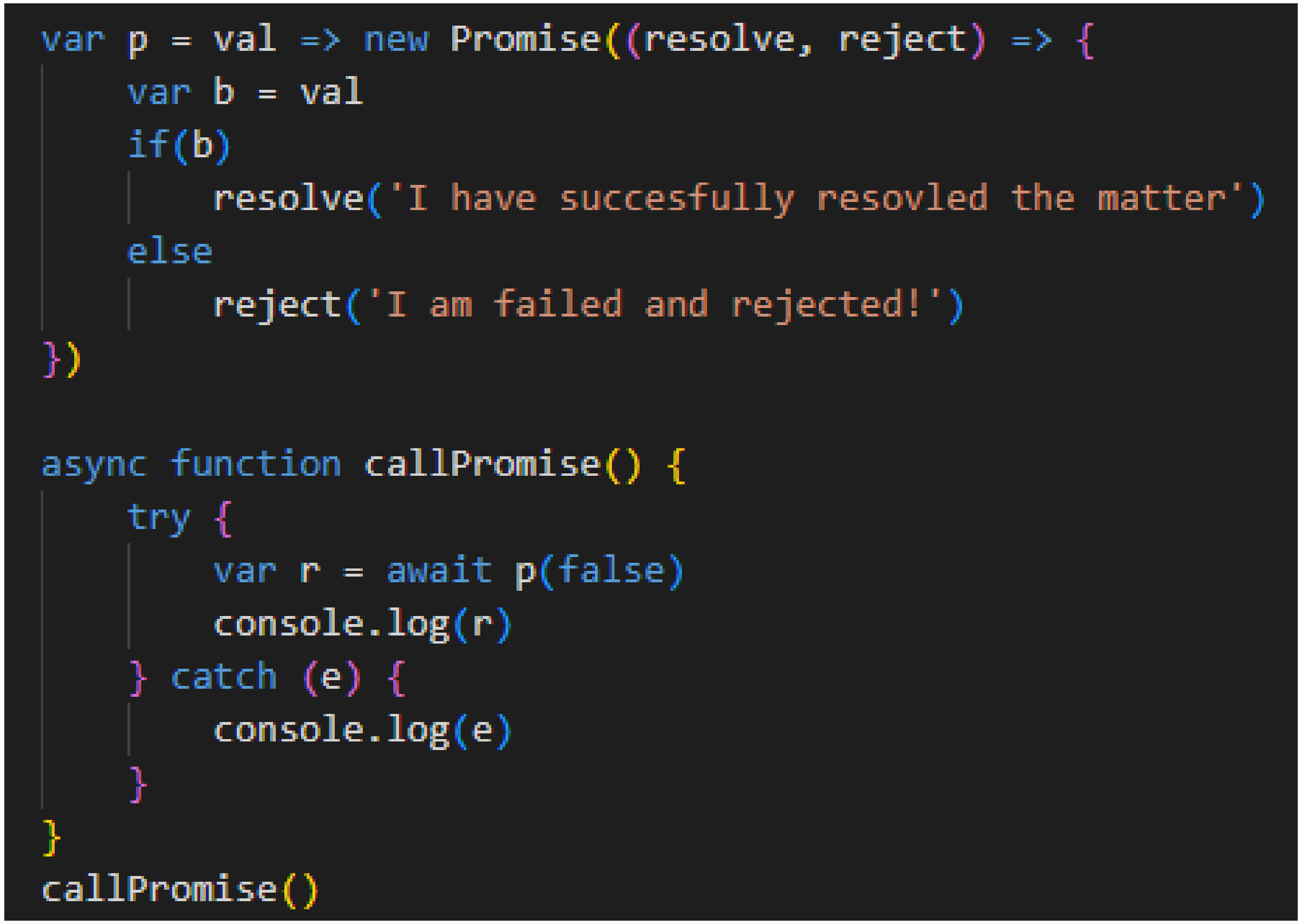
- then() returns a Promise based on the return value of the handler function
  - If `f2` return a regular value, the value becomes the result of the Promise
  - If `f2` return a promise, the result of that Promise becomes the result of the Promise returned by `then()`

… Understand Promising Chaining

```
f1.then(f2).then(f3)
```

- The result of `f1` is passed to `f2`
- The result of `f2` is passed to `f3`

Promise with async – await



```
var p = val => new Promise((resolve, reject) => {
    var b = val
    if(b)
        resolve('I have succesfully resovled the matter')
    else
        reject('I am failed and rejected!')
})

async function callPromise() {
    try {
        var r = await p(false)
        console.log(r)
    } catch (e) {
        console.log(e)
    }
}
callPromise()
```

Promise Example 2

- `write_file_promise.js`: open a file, add one line, then close the file
  - Use of promisify() in the Utilities package

Original function:

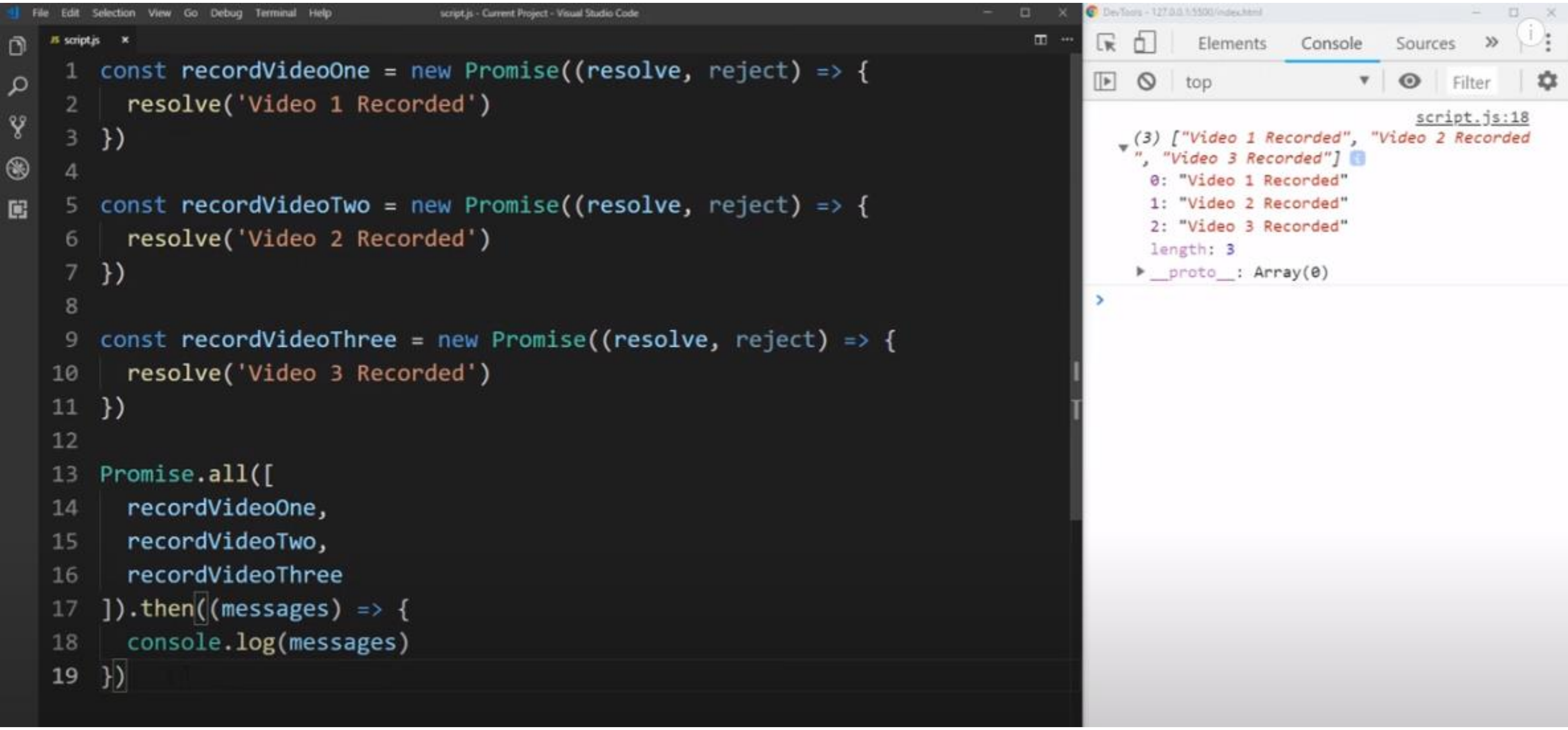
`func(args…, callback(err,result))`

Promisified:

`func(args…) returns a Promise`

```
"use strict";

const fs = require("fs");
const util = require("util");

const fopen = util.promisify(fs.open);
const fwrite = util.promisify(fs.write);
const fclose = util.promisify(fs.close);

let file = 0;

fopen("test.txt", "a")
  .then(fd => {
    file = fd;
    return fwrite(fd, "A New Line!\n");
  })
  .then(result => {
    console.log(`${result.bytesWritten} bytes written.`);
    return fclose(file);
  })
  .catch(err => console.log(err.message));
```

Promise



```javascript
const recordVideoOne = new Promise((resolve, reject) => {
  resolve('Video 1 Recorded')
})

const recordVideoTwo = new Promise((resolve, reject) => {
  resolve('Video 2 Recorded')
})

const recordVideoThree = new Promise((resolve, reject) => {
  resolve('Video 3 Recorded')
})

Promise.all([
  recordVideoOne,
  recordVideoTwo,
  recordVideoThree
]).then((messages) => {
  console.log(messages)
})
```

```
                                                   script.js:18
▼(3) ["Video 1 Recorded", "Video 2 Recorded
", "Video 3 Recorded"]
    0: "Video 1 Recorded"
    1: "Video 2 Recorded"
    2: "Video 3 Recorded"
    length: 3
  ▶__proto__: Array(0)
>
```

Parallel Execution

```
Promise.all([promise1, promise2 …]).then(
  function(results) {
    // results is an array of values, one
    // by each promise
  }
)

Promise.race([promise1, promise2 …]).then(
  function(result) {
    // result is the result of the promise
    // that settles first
  }
)
```

Running Node.js Server Applications

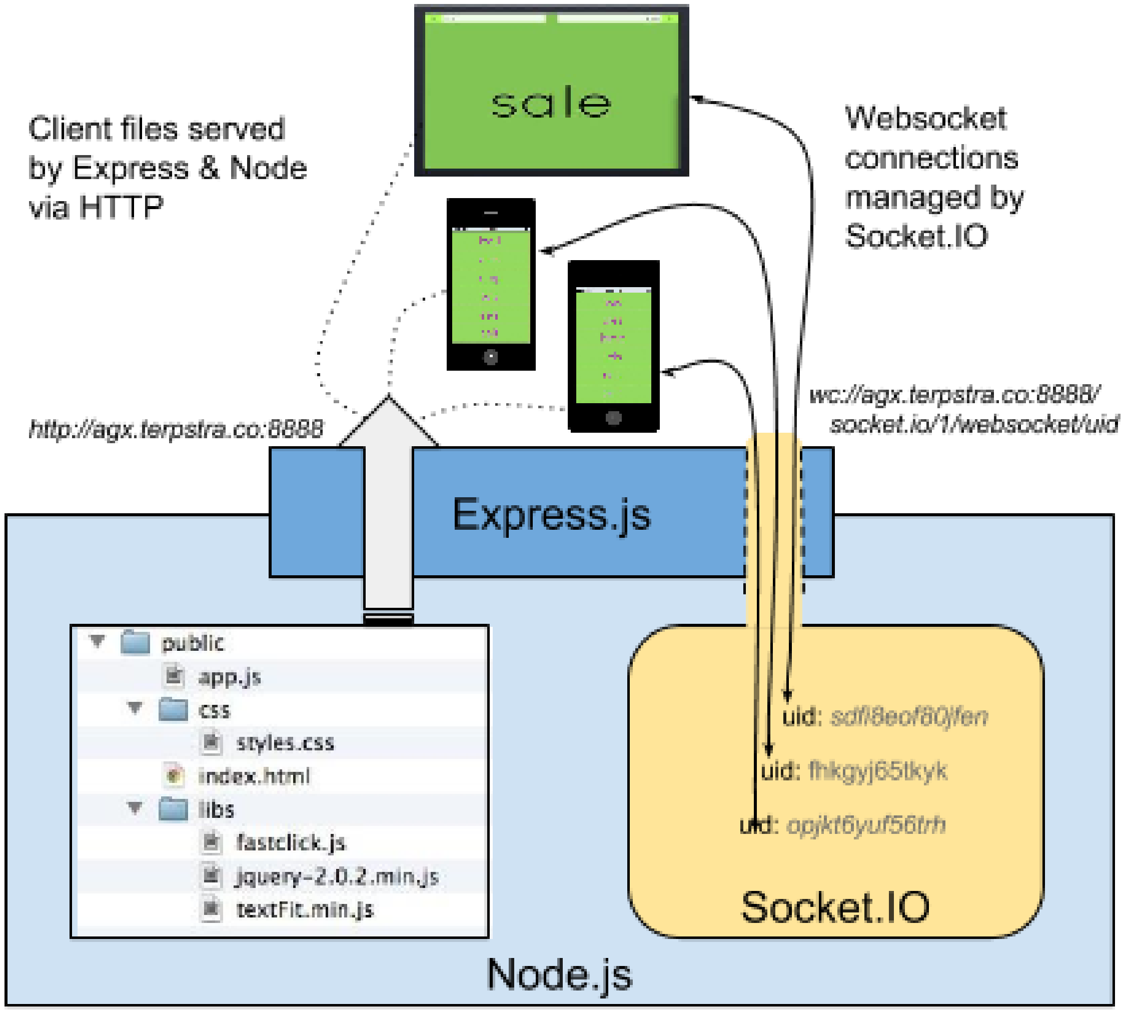
- Run server applications using nodemon during development
  - Automatically restart the application when changes in the project are detected
- Deploy server applications using pm2
  - Run server applications as managed background processes

Routing in Nodejs

```javascript
// Requiring the module
const http = require('http');

// Creating server object
const server = http.createServer((req, res) => {
    const url = req.url;
    if (url === '/') {
        res.write('<html>');
        res.write('<head><title>GeeksforGeeks</title><head>');
        res.write('<body><h2>Hello from Node.js server!!</h2></body>');
        res.write('</html>');
        return res.end();
    }
    if (url === '/about') {
        res.write('<html>');
        res.write('<head><title>GeeksforGeeks</title><head>');
        res.write('<body><h2>GeeksforGeeks- Node.js</h2></body>');
        res.write('</html>');
        return res.end();
    }
});

// Server setup
server.listen(3000, () => {
    console.log("Server listening on port 3000")
});
```

Express

- Minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.



Express gives ease of functionality

- Routing
- Delivery of Static Files
- “Middleware” – some ease in development (functionality)
- Form Processing
- Simple forms of Authentication
- View support
- Basic error handling, e.g. rejecting malformed requests

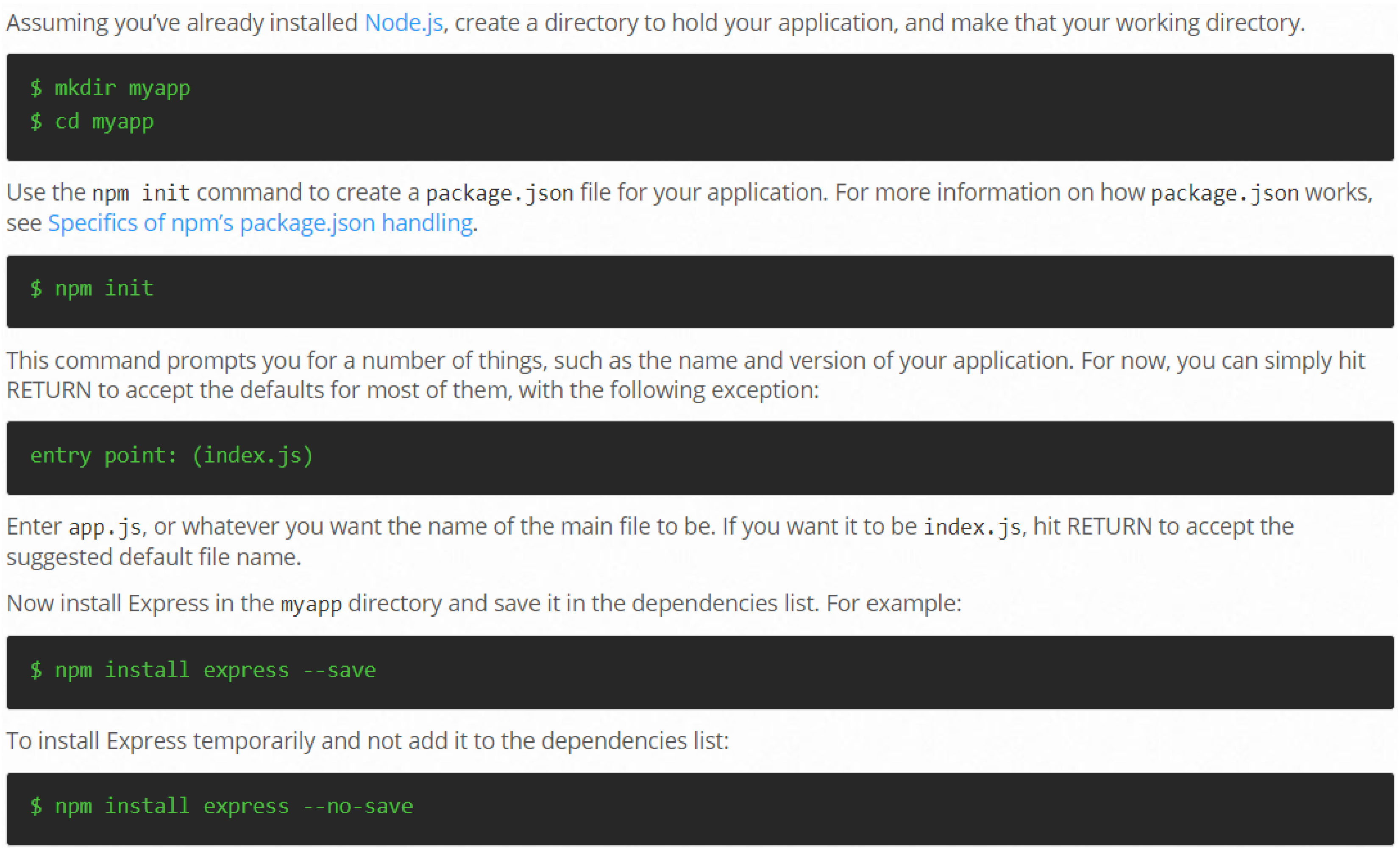
A lot of this you can do in NodeJS but, you may write more code to do it than if you use the framework Express.

There are other alternatives than Express (the **E** in M**E**AN) like Sail, Meteor

Express Basics

- Application
- Routing
- Handling requests
- Generating response
- Middleware
- Error handling

Express Installation



Assuming you've already installed Node.js, create a directory to hold your application, and make that your working directory.

```
$ mkdir myapp
$ cd myapp
```

Use the `npm init` command to create a `package.json` file for your application. For more information on how `package.json` works, see Specifics of npm's package.json handling.

```
$ npm init
```

This command prompts you for a number of things, such as the name and version of your application. For now, you can simply hit RETURN to accept the defaults for most of them, with the following exception:

```
entry point: (index.js)
```

Enter `app.js`, or whatever you want the name of the main file to be. If you want it to be `index.js`, hit RETURN to accept the suggested default file name.

Now install Express in the `myapp` directory and save it in the dependencies list. For example:

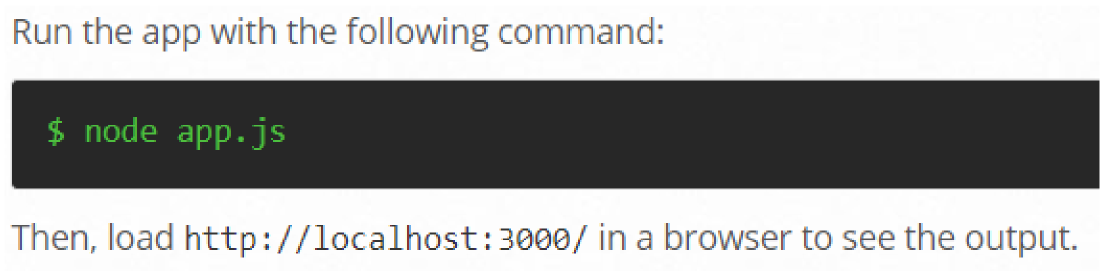
```
$ npm install express --save
```

To install Express temporarily and not add it to the dependencies list:

```
$ npm install express --no-save
```

HelloWorld in Express

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

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

app.listen(3000, () =>
  console.log('Listening on port 3000'));
```


Run the app with the following command:

```
$ node app.js
```

Then, load `http://localhost:3000/` in a browser to see the output.

Application

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

- The Application object
  - Routing requests
  - Rendering views
  - Configuring middleware

Routing Methods in App

- `app.all( path, callback [,callback …])`

```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...')
  next() // pass control to the next handler
})
```

```
app.all('*', requireAuthentication, loadUser)
```

```
app.all('/api/*', requireAuthentication)
```

- `app.METHOD( path, callback, [,callback …])`
  - `METHOD` is one of the routing methods, e.g. `get`, `post`, and so on

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

Respond to POST request on the root route (/), the application's home page:

```
app.post('/', function (req, res) {
  res.send('Got a POST request')
})
```

Respond to a PUT request to the /user route:

```
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user')
})
```

Respond to a DELETE request to the /user route:

```
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user')
})
```

National University
Of Computer and Emerging Sciences

Modularize Endpoints Using Express Router …

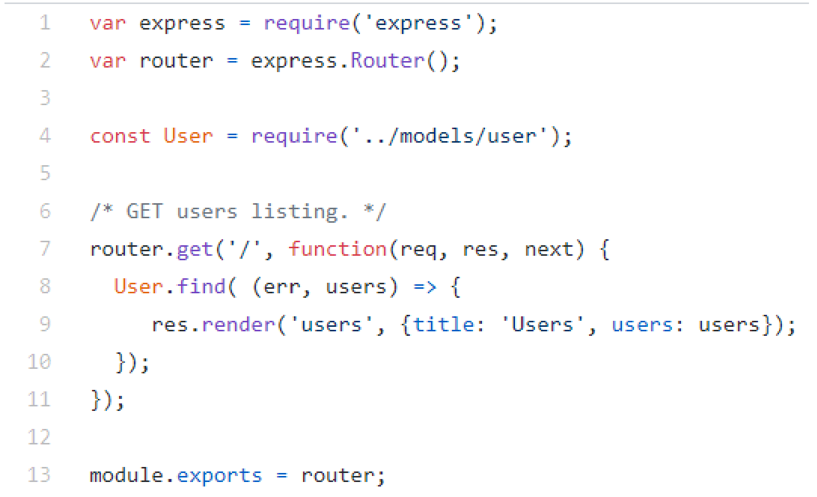
- Example
  - List users: `/users/`, `GET`
  - Add user: `/users/`, `POST`
  - Get user: `/users/:id`, `GET`
  - Delete user: `/users/:id`, `DELETE`

… Modularize Endpoints Using Express Router

```
const router = express.Router();

router.get('/', …);
router.post('/', …);
… …
module.exports = router;
```

- A router is like a "mini app"

users.js



```
var express = require('express');
var router = express.Router();

const User = require('../models/user');

/* GET users listing. */
router.get('/', function(req, res, next) {
  User.find( (err, users) => {
      res.render('users', {title: 'Users', users: users});
  });
});

module.exports = router;
```

… Modularize Endpoints Using Express Router

```
const users = require('./users');
app.use('/users', users);
```

- Attach the router to the main app at the URL

app.js

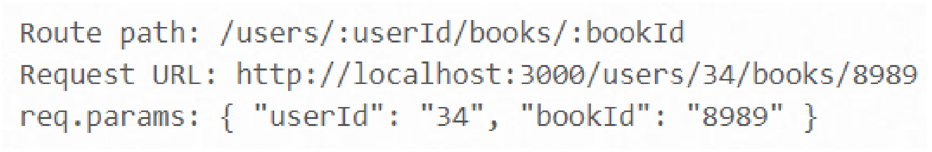
```
app.use('/', indexController);
app.use('/users', usersController);
app.use('/api/login', loginRestController);
```

Handling Requests

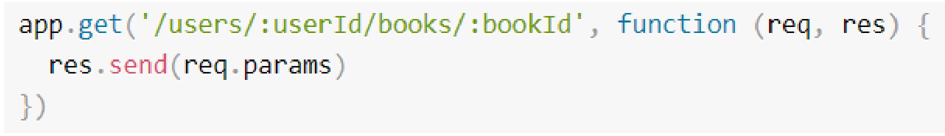
- Request
  - Properties for basic request information such as URL, method, cookies
  - Get header: get()
  - User input
    - Request parameters: req.query
    - Route parameters: req.params
    - Form data: req.body
    - JSON data: req.body

Example: Add

- GET: `/add?a=10&b=20`
- GET: `/add/a/10/b/20`
- POST (Form): `/add`
  - Body: `a=10&b=20`
- POST (JSON): `/add`
  - Content-Type: `application/json`
  - Body: `{"a": 10, "b": 20}`



```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```



```
app.get('/users/:userId/books/:bookId', function (req, res) {
  res.send(req.params)
})
```

Generating Response

- Response
  - Set status: status()
    - end()
  - Send JSON: json()
  - Send other data: send()
  - Redirect: redirect()
  - Other methods for set headers, cookies, download files etc.

```
res
  .status(201)
  .cookie('access_token', 'Bearer ' + token, {
    expires: new Date(Date.now() + 8 * 3600000) // cookie will be removed after 8 hours
  })
  .cookie('test', 'test')
  .redirect(301, '/admin')
```

```
res.json(null)
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```

```
app.get('/', function (req, res) {
  console.dir(res.headersSent) // false
  res.send('OK')
  console.dir(res.headersSent) // true
})
```

```
res.redirect('/foo/bar')
res.redirect('http://example.com')
res.redirect(301, 'http://example.com')
res.redirect('../login')
```

Middleware for Express

A software with functions that have access to: Request Object | Response Object

Executes during the request and the response cycle

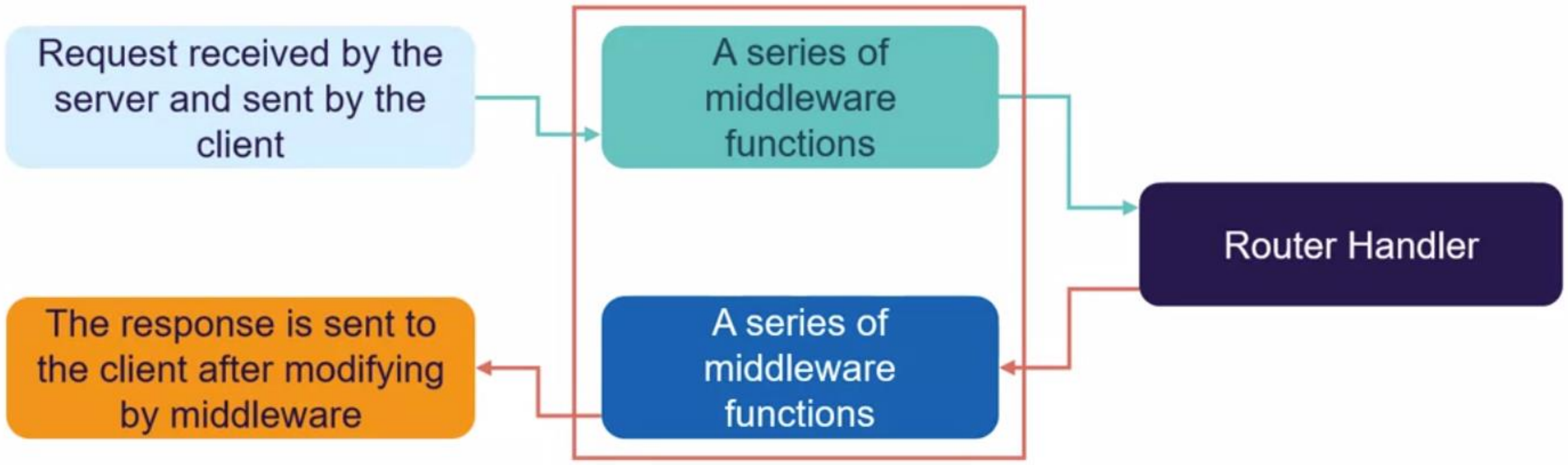
Can be used for:

Logger	Authentication	Parsing JSON Data
▪ Logs user information	▪ Protects the routes	▪ Executes any code ▪ Makes changes to request and response objects ▪ Ends the request-response cycle ▪ Calls the next middleware in the stack

Express middleware includes application-level, router-level, and error handling functionalities.

It can be built in or extracted from a third-party module.

Middleware for Express

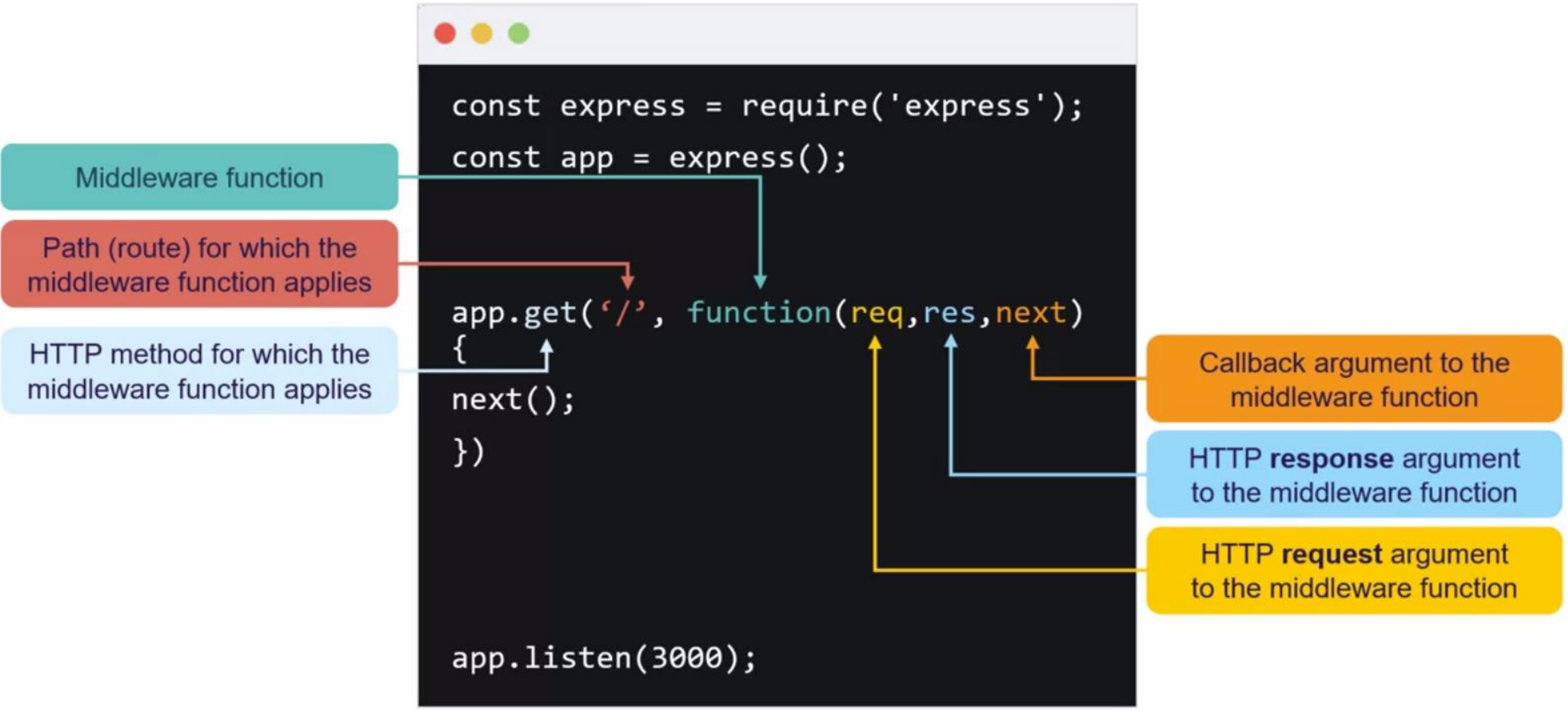

Request received by the server and sent by the client
A series of middleware functions
Router Handler
A series of middleware functions
The response is sent to the client after modifying by middleware

next() is a callback function that passes control to the next middleware function.

The chain ends if the next() method in the series of middleware is not called.

The request will be left hanging if the request-response cycle does not end.

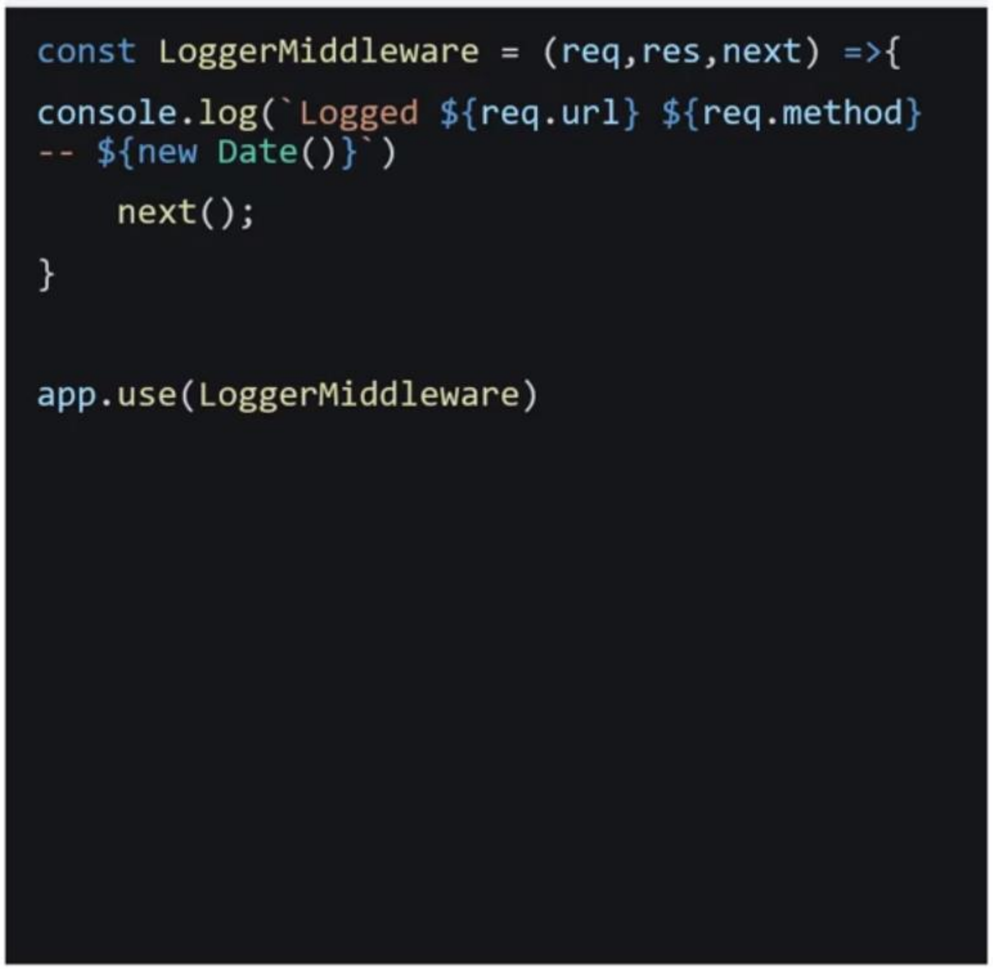
Middleware for Express



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


app.get('/', function(req,res,next)
{
next();
})



app.listen(3000);
```

Middleware function
Path (route) for which the middleware function applies
HTTP method for which the middleware function applies
Callback argument to the middleware function
HTTP **response** argument to the middleware function
HTTP **request** argument to the middleware function

Global Middleware

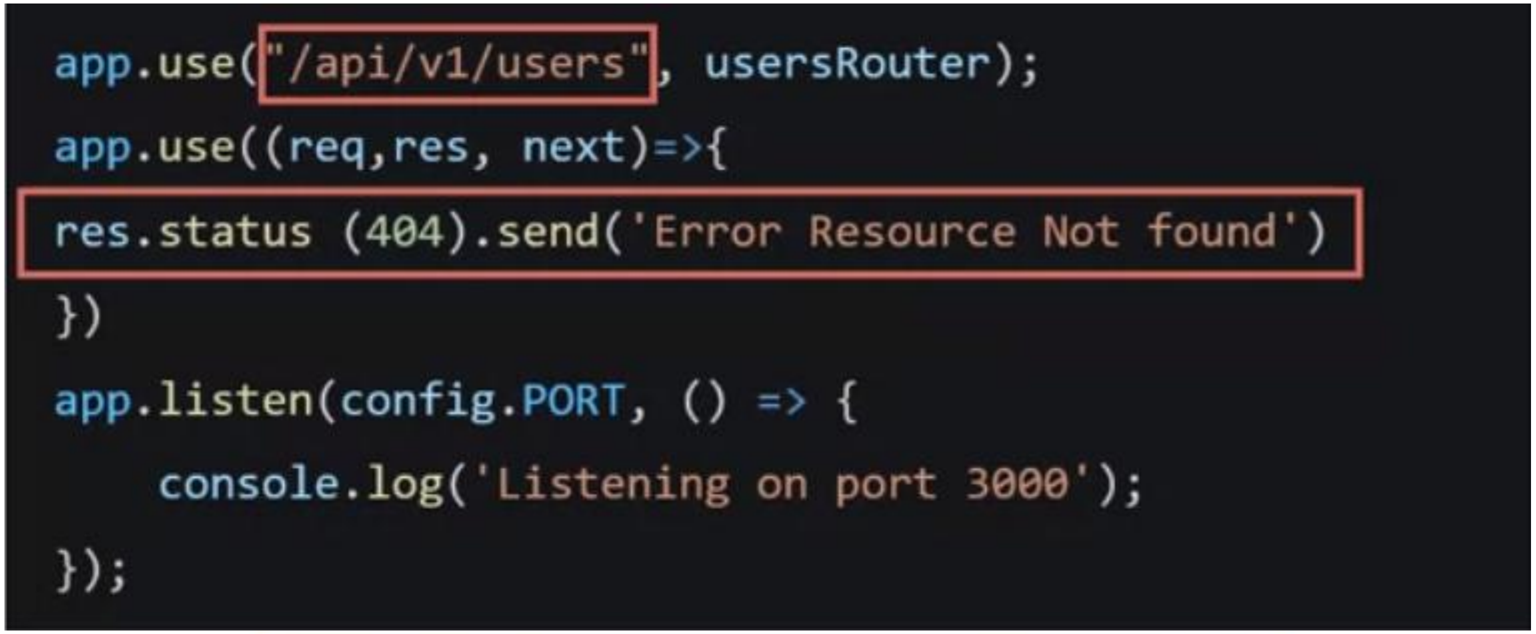


```
const LoggerMiddleware = (req,res,next) =>{
console.log(`Logged ${req.url} ${req.method}
-- ${new Date()}`)
    next();
}


app.use(LoggerMiddleware)
```

Middleware Function—Logger

- Helps trace the errors of the application
- Helps in creating custom loggers
- Takes three parameters:
  - request
  - response
  - next()
- Requires the `app.use()` function to load

```
Logged  /  GET --
Mon Nov 29, 2021 19:10:53 GMT+0530
(India Standard Time)
```



```
app.use("/api/v1/users", usersRouter);
app.use((req,res, next)=>{
res.status (404).send('Error Resource Not found')
})
app.listen(config.PORT, () => {
    console.log('Listening on port 3000');
});
```

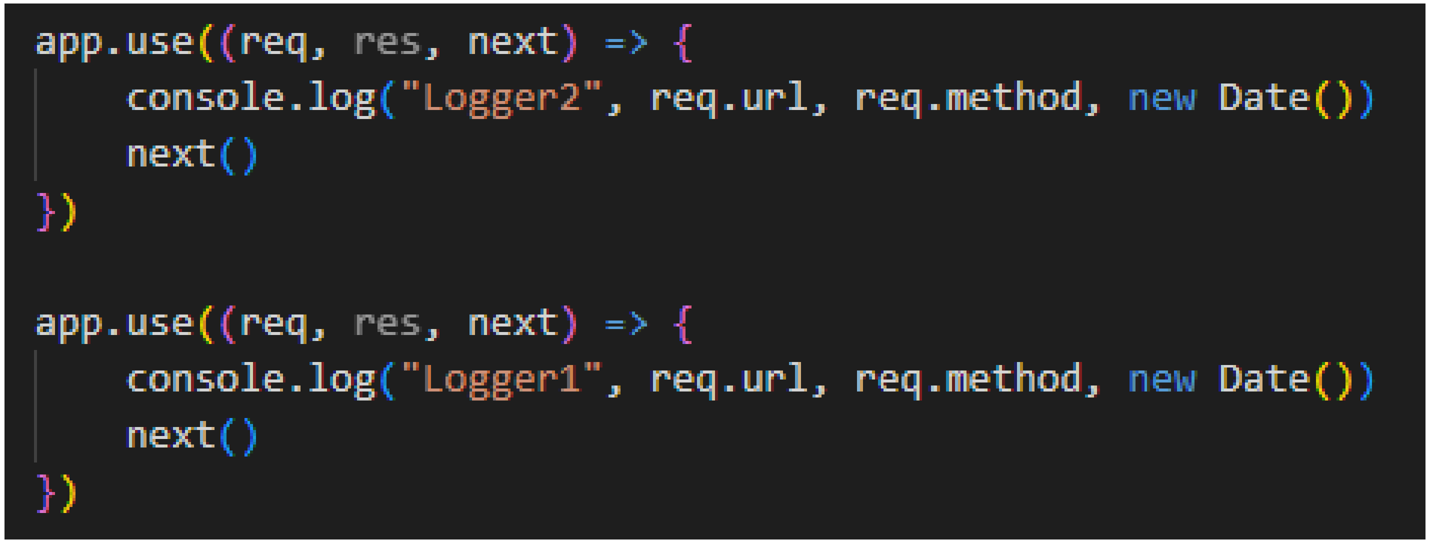
Middleware Function—Error

- Called if the specified route is not present
- Use status code 404 and message as "Error Resource Not Found"
- Loggers have to be called before the routes and error has to be called after the routes
- Loaded by the `app.use()` function

National University
Of Computer and Emerging Sciences

Global Middleware

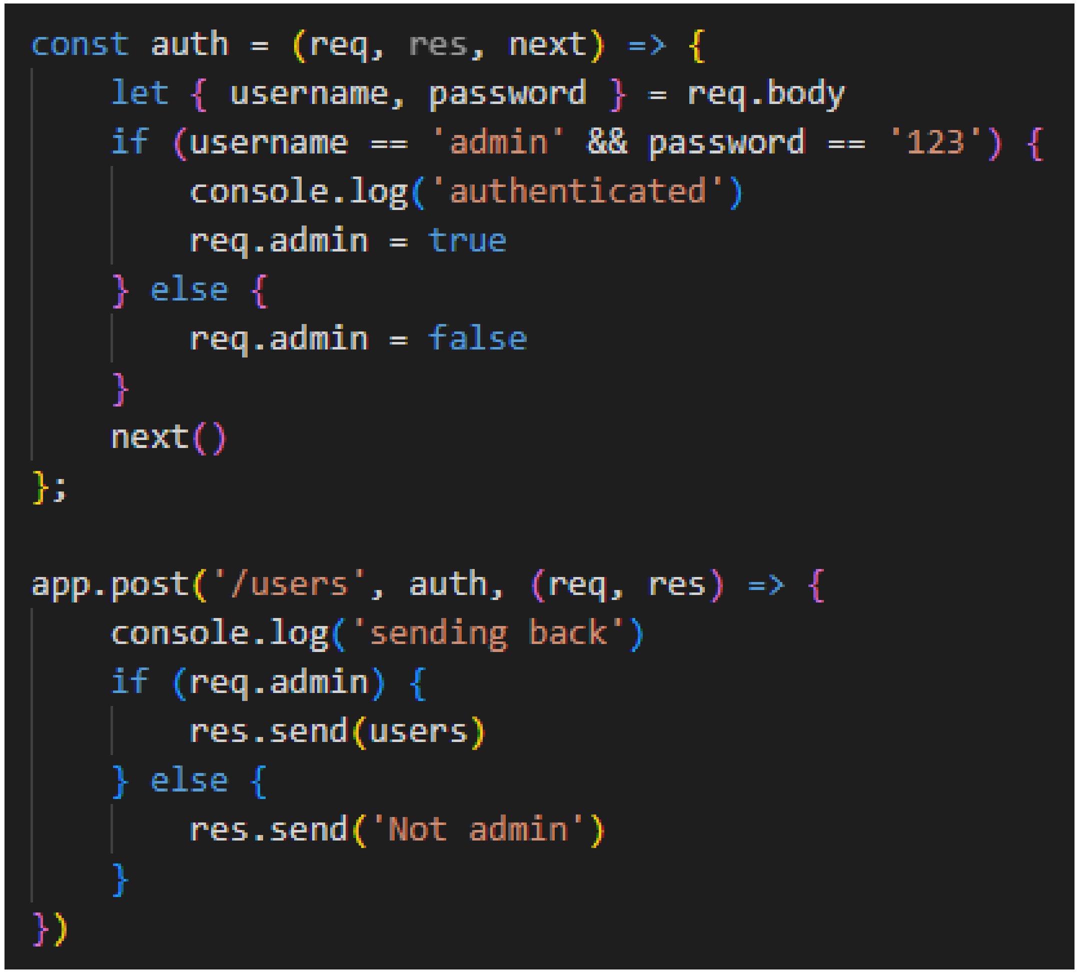
- Executes in an order
- Executes on every request



```
app.use((req, res, next) => {
    console.log("Logger2", req.url, req.method, new Date())
    next()
})

app.use((req, res, next) => {
    console.log("Logger1", req.url, req.method, new Date())
    next()
})
```

Route Specific Middleware

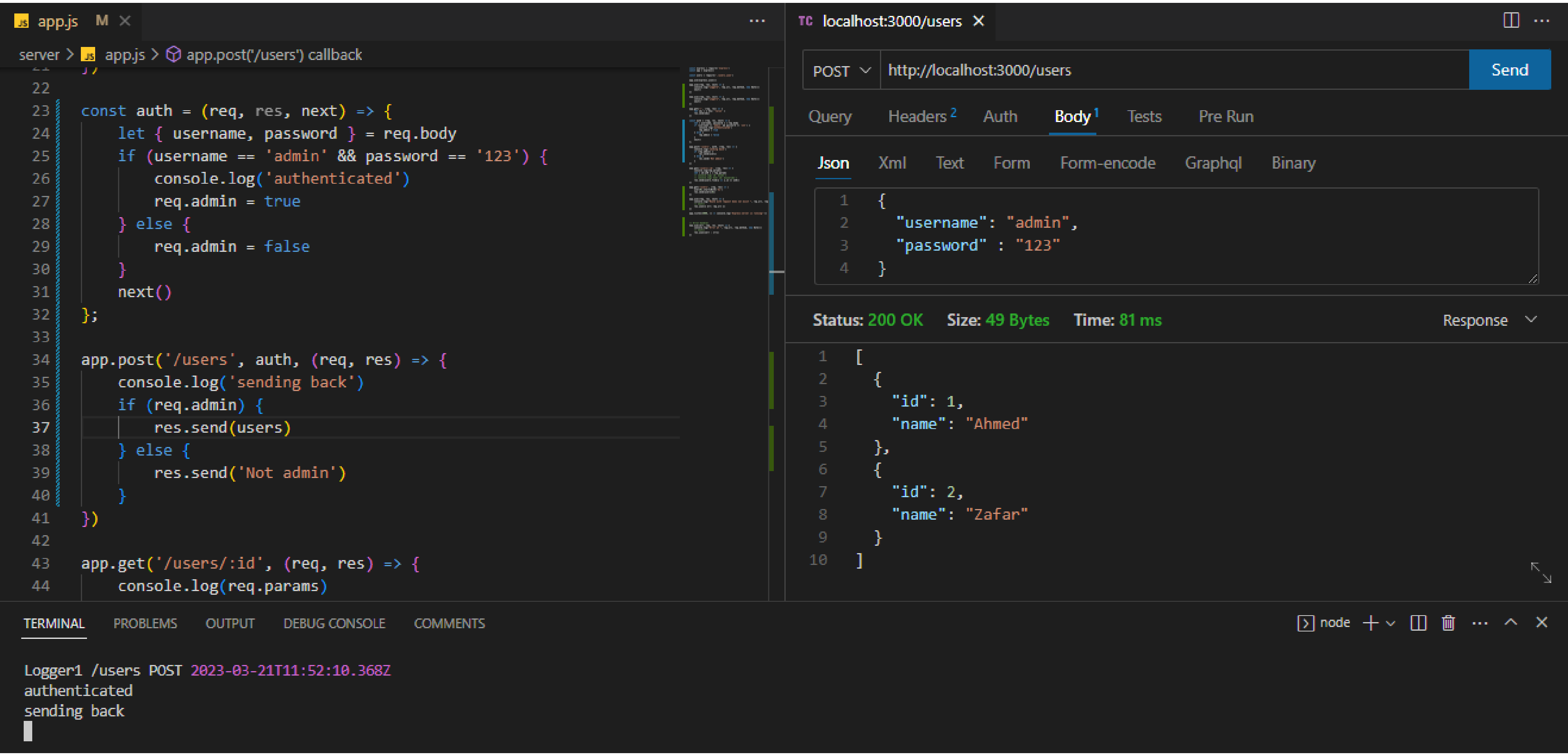
- Auth middleware will be called when a POST request is sent on ‘/users’ route
- We can add properties into **req** object and can access in next middleware



```
const auth = (req, res, next) => {
    let { username, password } = req.body
    if (username == 'admin' && password == '123') {
        console.log('authenticated')
        req.admin = true
    } else {
        req.admin = false
    }
    next()
};

app.post('/users', auth, (req, res) => {
    console.log('sending back')
    if (req.admin) {
        res.send(users)
    } else {
        res.send('Not admin')
    }
})
```

National University
Of Computer and Emerging Sciences

Sample Request/Response (success)



```javascript
const auth = (req, res, next) => {
    let { username, password } = req.body
    if (username == 'admin' && password == '123') {
        console.log('authenticated')
        req.admin = true
    } else {
        req.admin = false
    }
    next()
};

app.post('/users', auth, (req, res) => {
    console.log('sending back')
    if (req.admin) {
        res.send(users)
    } else {
        res.send('Not admin')
    }
})

app.get('/users/:id', (req, res) => {
    console.log(req.params)
```

POST http://localhost:3000/users

```json
{
  "username": "admin",
  "password" : "123"
}
```

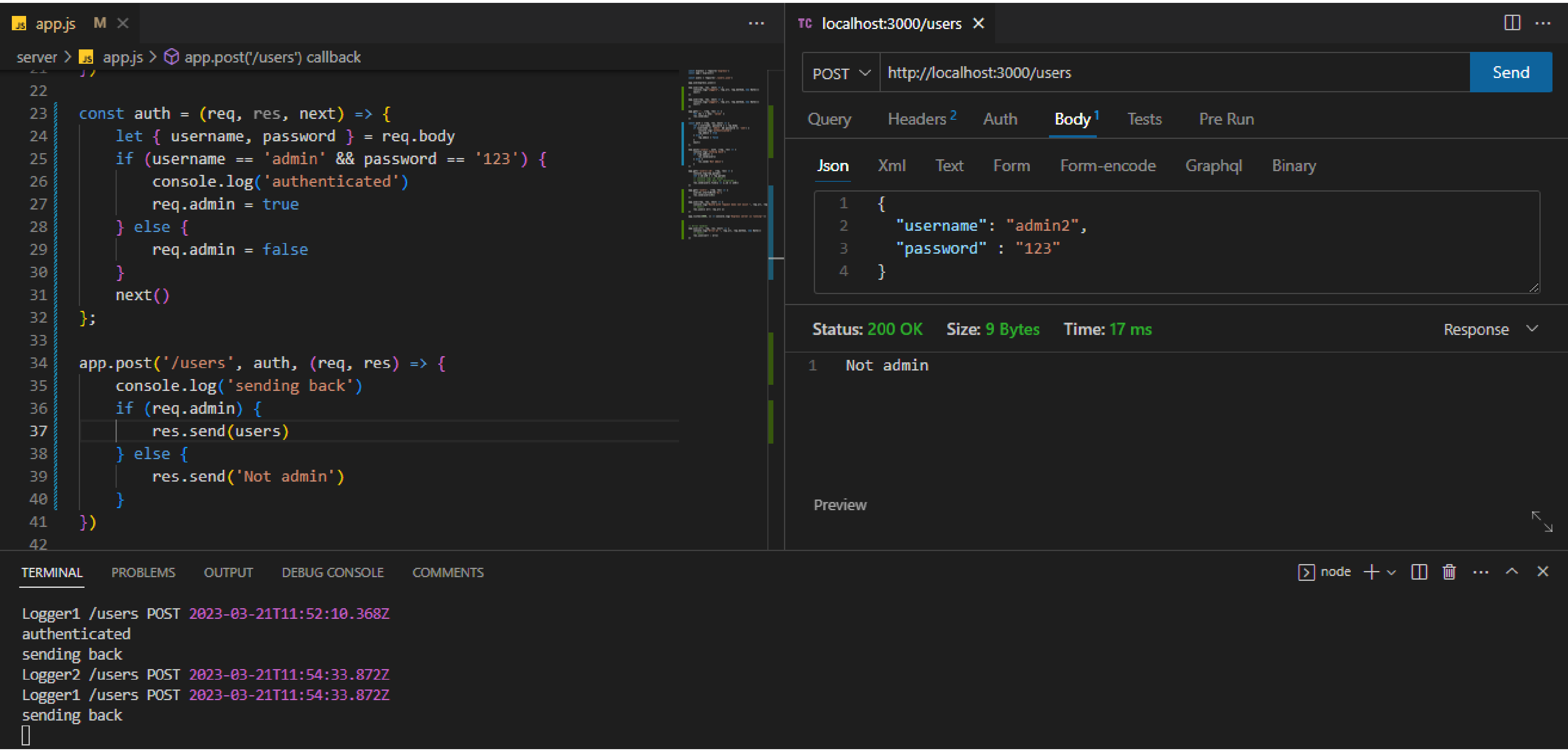
Status: 200 OK Size: 49 Bytes Time: 81 ms

```json
[
  {
    "id": 1,
    "name": "Ahmed"
  },
  {
    "id": 2,
    "name": "Zafar"
  }
]
```

```
Logger1 /users POST 2023-03-21T11:52:10.368Z
authenticated
sending back
```

National University
Of Computer and Emerging Sciences

Sample Request/Response (fail)



```javascript
const auth = (req, res, next) => {
    let { username, password } = req.body
    if (username == 'admin' && password == '123') {
        console.log('authenticated')
        req.admin = true
    } else {
        req.admin = false
    }
    next()
};

app.post('/users', auth, (req, res) => {
    console.log('sending back')
    if (req.admin) {
        res.send(users)
    } else {
        res.send('Not admin')
    }
})
```

POST http://localhost:3000/users Send

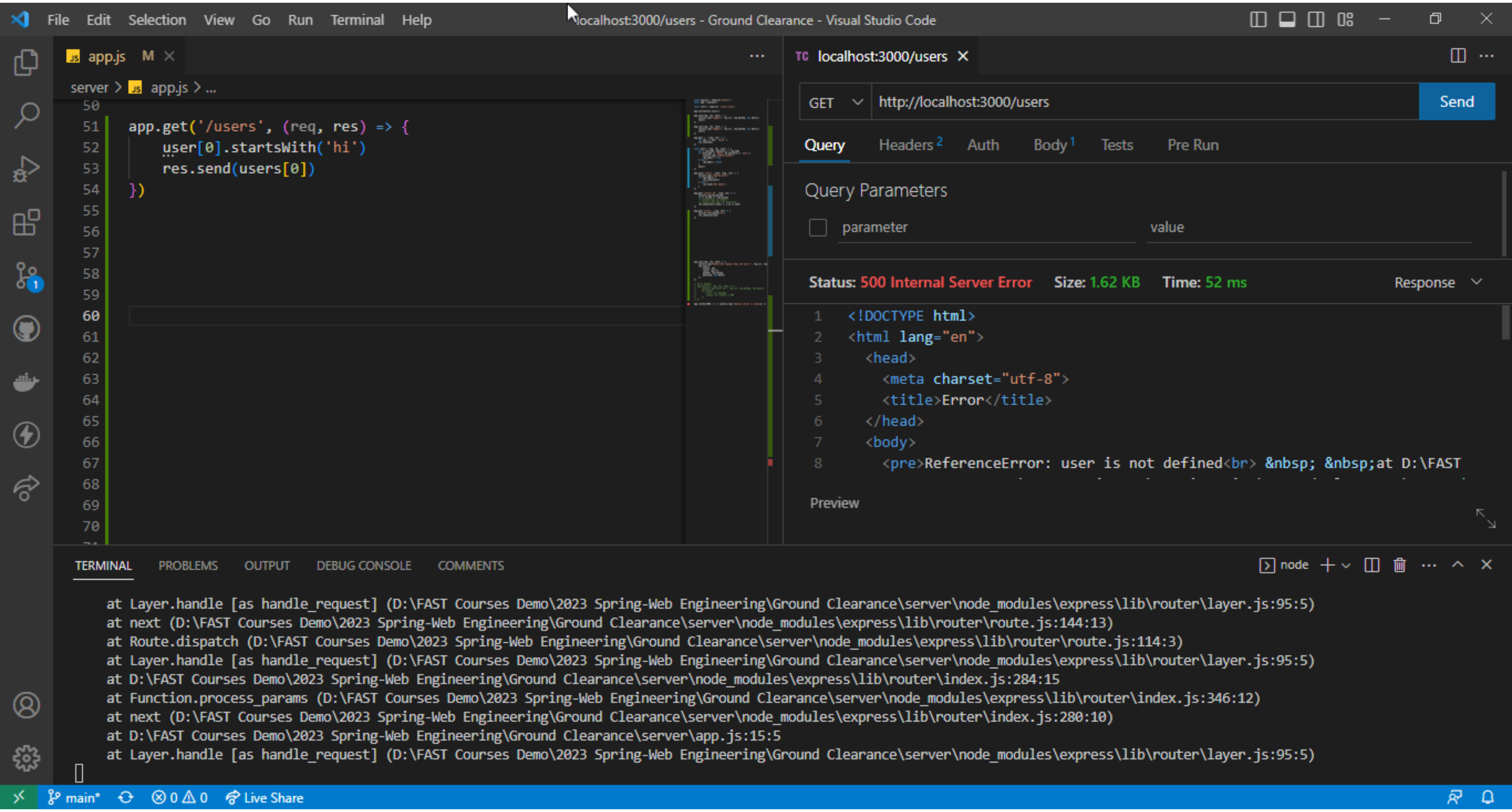
```json
{
  "username": "admin2",
  "password" : "123"
}
```

Status: 200 OK Size: 9 Bytes Time: 17 ms

```
Not admin
```

```
Logger1 /users POST 2023-03-21T11:52:10.368Z
authenticated
sending back
Logger2 /users POST 2023-03-21T11:54:33.872Z
Logger1 /users POST 2023-03-21T11:54:33.872Z
sending back
```

App Crashes on Error



```js
app.get('/users', (req, res) => {
    user[0].startsWith('hi')
    res.send(users[0])
})
```

GET http://localhost:3000/users

Status: 500 Internal Server Error   Size: 1.62 KB   Time: 52 ms

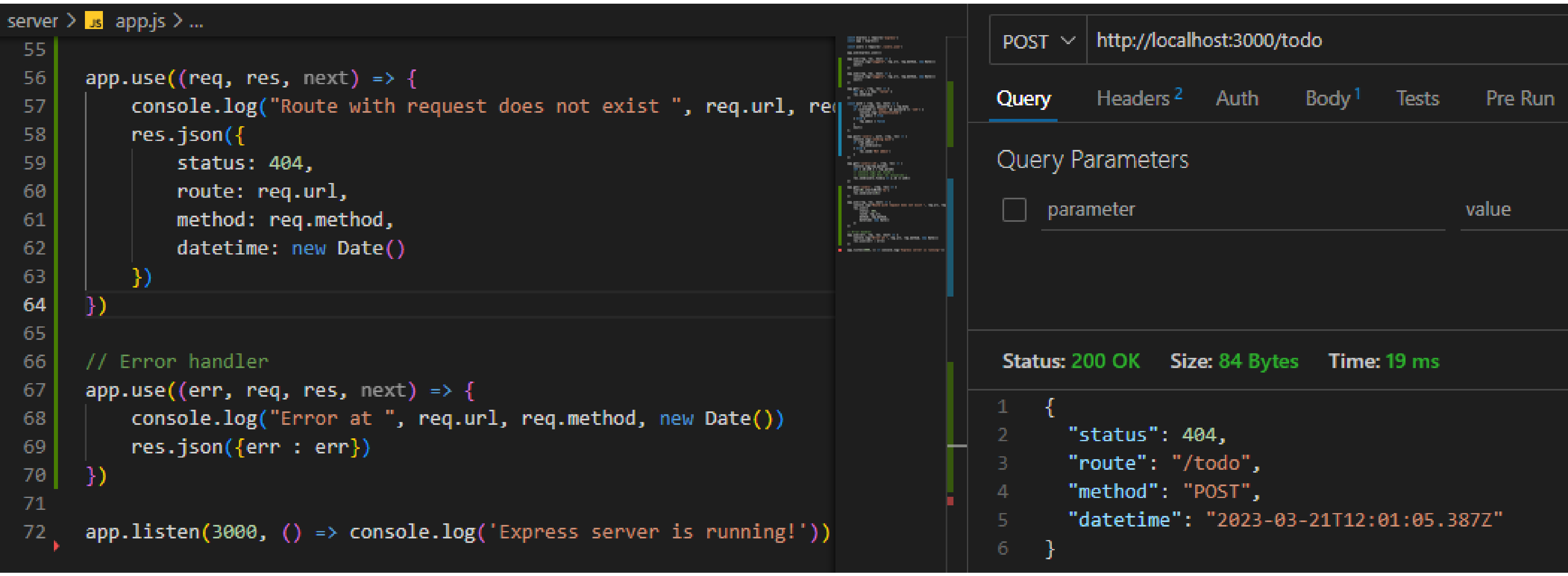
```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Error</title>
  </head>
  <body>
    <pre>ReferenceError: user is not defined<br> &nbsp; &nbsp;at D:\FAST
```

```
    at Layer.handle [as handle_request] (D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\node_modules\express\lib\router\layer.js:95:5)
    at next (D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\node_modules\express\lib\router\route.js:144:13)
    at Route.dispatch (D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\node_modules\express\lib\router\route.js:114:3)
    at Layer.handle [as handle_request] (D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\node_modules\express\lib\router\layer.js:95:5)
    at D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\node_modules\express\lib\router\index.js:284:15
    at Function.process_params (D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\node_modules\express\lib\router\index.js:346:12)
    at next (D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\node_modules\express\lib\router\index.js:280:10)
    at D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\app.js:15:5
    at Layer.handle [as handle_request] (D:\FAST Courses Demo\2023 Spring-Web Engineering\Ground Clearance\server\node_modules\express\lib\router\layer.js:95:5)
```

File Edit Selection View Go Run Terminal Help
localhost:3000/users - Ground Clearance - Visual Studio Code
app.js M
server > app.js > ...
localhost:3000/users
Send
Query Headers 2 Auth Body 1 Tests Pre Run
Query Parameters
parameter
value
Response
Preview
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE COMMENTS
node
main*
0 0
Live Share

Avoid Default Error Page and Error Handling Middleware

- To avoid default error page
- To avoid app crashing
- Add these middleware at the end of all requests



server > app.js > ...

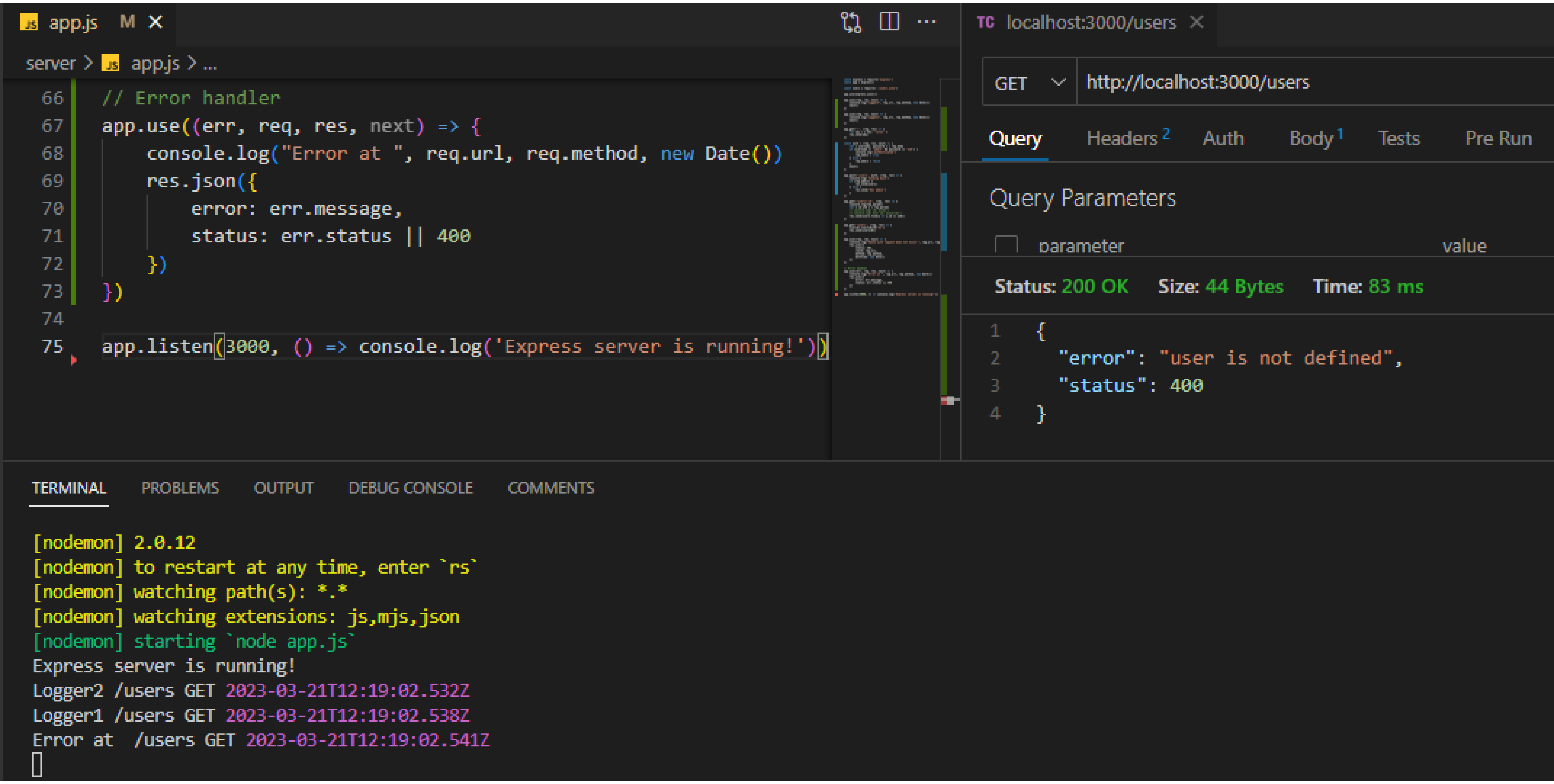
```javascript

app.use((req, res, next) => {
    console.log("Route with request does not exist ", req.url, re
    res.json({
        status: 404,
        route: req.url,
        method: req.method,
        datetime: new Date()
    })
})

// Error handler
app.use((err, req, res, next) => {
    console.log("Error at ", req.url, req.method, new Date())
    res.json({err : err})
})

app.listen(3000, () => console.log('Express server is running!'))
```

POST http://localhost:3000/todo

Status: 200 OK Size: 84 Bytes Time: 19 ms

```json
{
  "status": 404,
  "route": "/todo",
  "method": "POST",
  "datetime": "2023-03-21T12:01:05.387Z"
}
```

Error Handling Middleware



```javascript
// Error handler
app.use((err, req, res, next) => {
    console.log("Error at ", req.url, req.method, new Date())
    res.json({
        error: err.message,
        status: err.status || 400
    })
})

app.listen(3000, () => console.log('Express server is running!'))
```

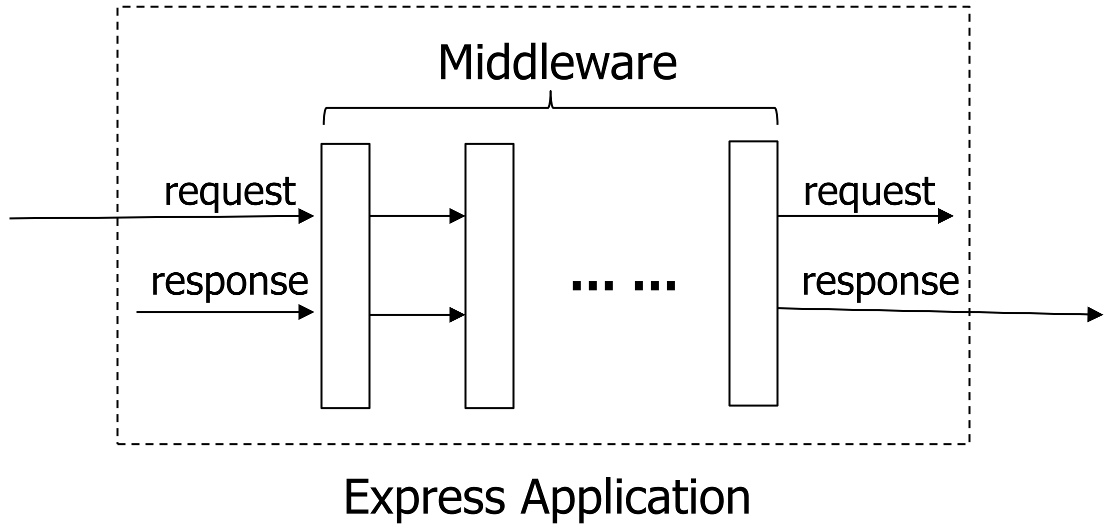
Status: 200 OK   Size: 44 Bytes   Time: 83 ms

```json
{
  "error": "user is not defined",
  "status": 400
}
```

```
[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
Express server is running!
Logger2 /users GET 2023-03-21T12:19:02.532Z
Logger1 /users GET 2023-03-21T12:19:02.538Z
Error at  /users GET 2023-03-21T12:19:02.541Z
```

National University
Of Computer and Emerging Sciences

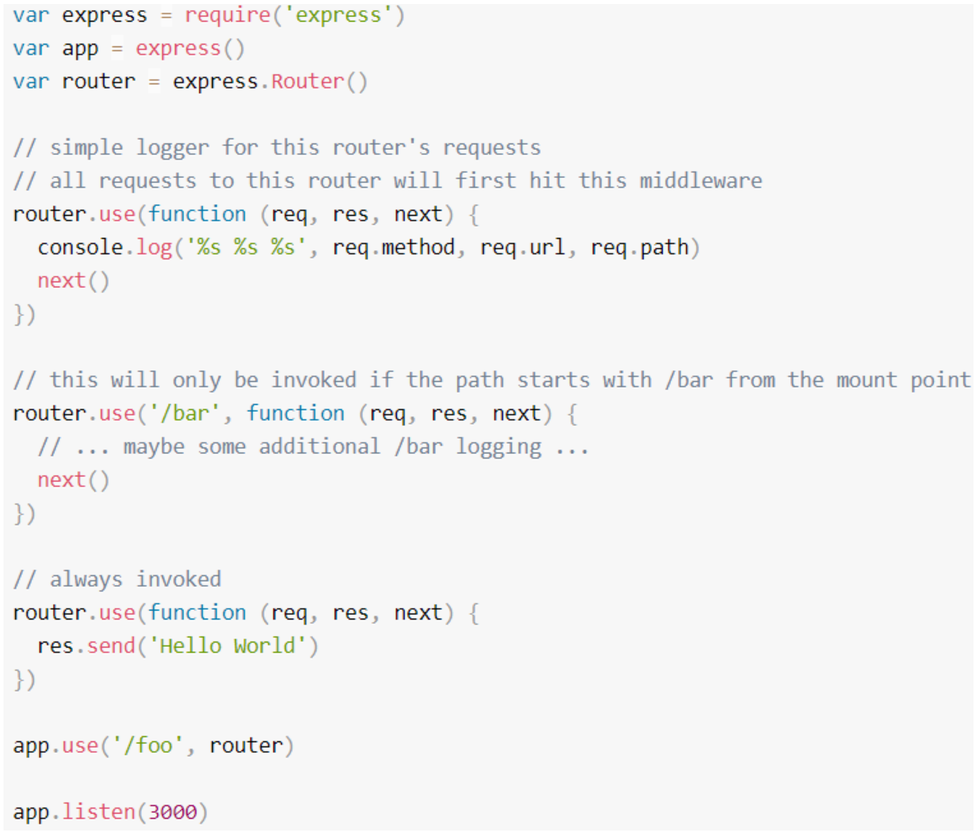
Middleware



- A middleware is a function that has access to three arguments: the `request` object, the `response` object, and a `next` function that passes control to the next middleware function

Middleware Example

- Create a simple request logger middleware that prints out request URL, method, and time
  - The `next` argument
  - Add the middleware to the application using `app.use()`
  - Middleware can also be added at the router level with `router.use()`



```javascript
var express = require('express')
var app = express()
var router = express.Router()

// simple logger for this router's requests
// all requests to this router will first hit this middleware
router.use(function (req, res, next) {
  console.log('%s %s %s', req.method, req.url, req.path)
  next()
})

// this will only be invoked if the path starts with /bar from the mount point
router.use('/bar', function (req, res, next) {
  // ... maybe some additional /bar logging ...
  next()
})

// always invoked
router.use(function (req, res, next) {
  res.send('Hello World')
})

app.use('/foo', router)

app.listen(3000)
```

Other Middlewares

- `express.json()` parses JSON request body and add JSON object properties to `req.body`
- `express.urlencoded()` parses urlencoded request body and request parameters to `req.body`
- Route handler functions are also middleware
  - Where is `next`??
  - Remember to use `next` if you have more than one handler functions for a route
- *Middleware order is important!*

Error Handling Middleware

- Error handling middleware has an extra argument `err`, e.g. `(err, req, res, next)`
- Calling `next(err)` will bypass the rest of the regular middleware and pass control to the next error handling middleware
  - `err` is an Error object
- Express adds a default error handling middleware at the end of the middleware chain

Error Handling Example

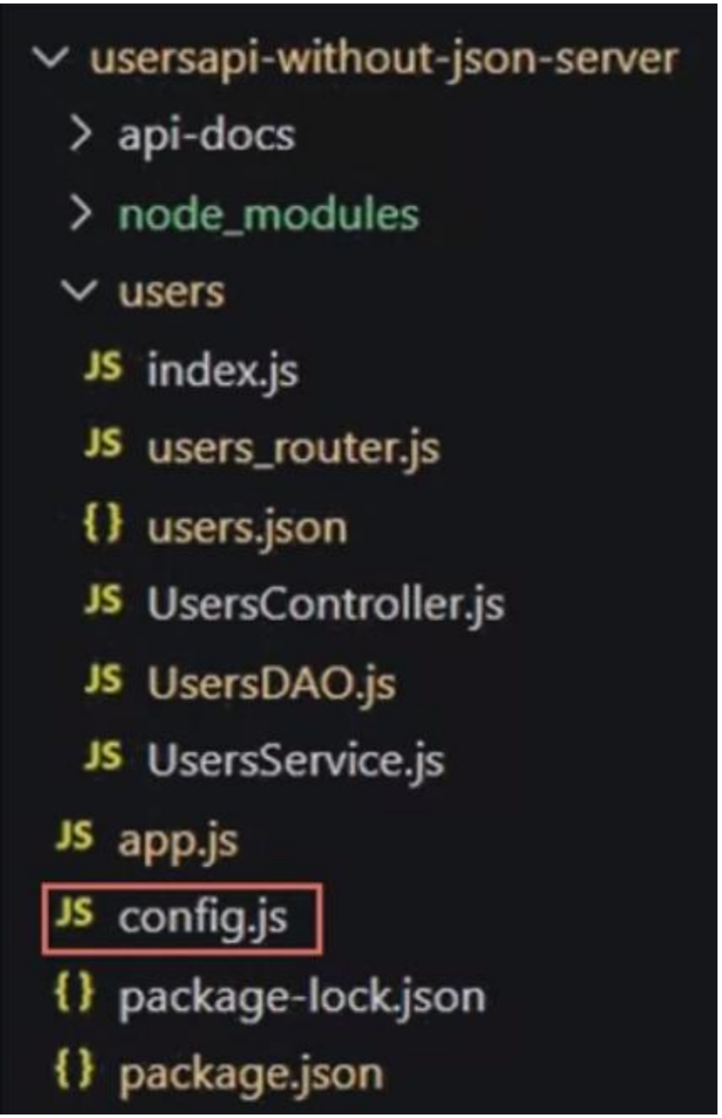
- Create a middleware that handles API errors, i.e. returns JSON instead of an error page

```
// error handler
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  if (req.originalUrl.startsWith('/api/')) {
    res.json({ msg: err.message });
  } else {
    res.locals.message = err.message;
    res.locals.error = req.app.get('env') === 'development' ? err : {};
    res.render('error');
  }
});
```

Nodejs Application Structure

Routes	• Forward the request to appropriate controller functions • To make the code modular, use the command: ◦ `const express = require('express');` ◦ `const router = express.Router();` • Route handlers can be defined separately in a `.js` file instead of an `app.js` file.
Controller	Callback functions passed to the router methods
Service Layer	Handles the business logic of the application
DAO Layer	Used to perform operations on the data resource

Nodejs Application Structure



```
∨ usersapi-without-json-server
  > api-docs
  > node_modules
  ∨ users
    JS index.js
    JS users_router.js
    {} users.json
    JS UsersController.js
    JS UsersDAO.js
    JS UsersService.js
  JS app.js
  JS config.js
  {} package-lock.json
  {} package.json
```

`app.js` is the entry point for the application and calls `index.js` for the routes.

The `index.js` file references the `users_router.js`.

The `users_router.js` file contains all the routes.

`users.json` consists of data about the users.

`UsersDAO.js` performs all manipulation operations on the data.

`UsersService.js` contains code to perform all the business logic.

`UsersController.js` handles incoming requests and returns responses.

`config.js` consists of configuration details.

Readings

- Express Documentation

Thank you!