

# Backend Development– W1S2

## Express.js

Cyrille Jegourel – Singapore University of Technology and Design



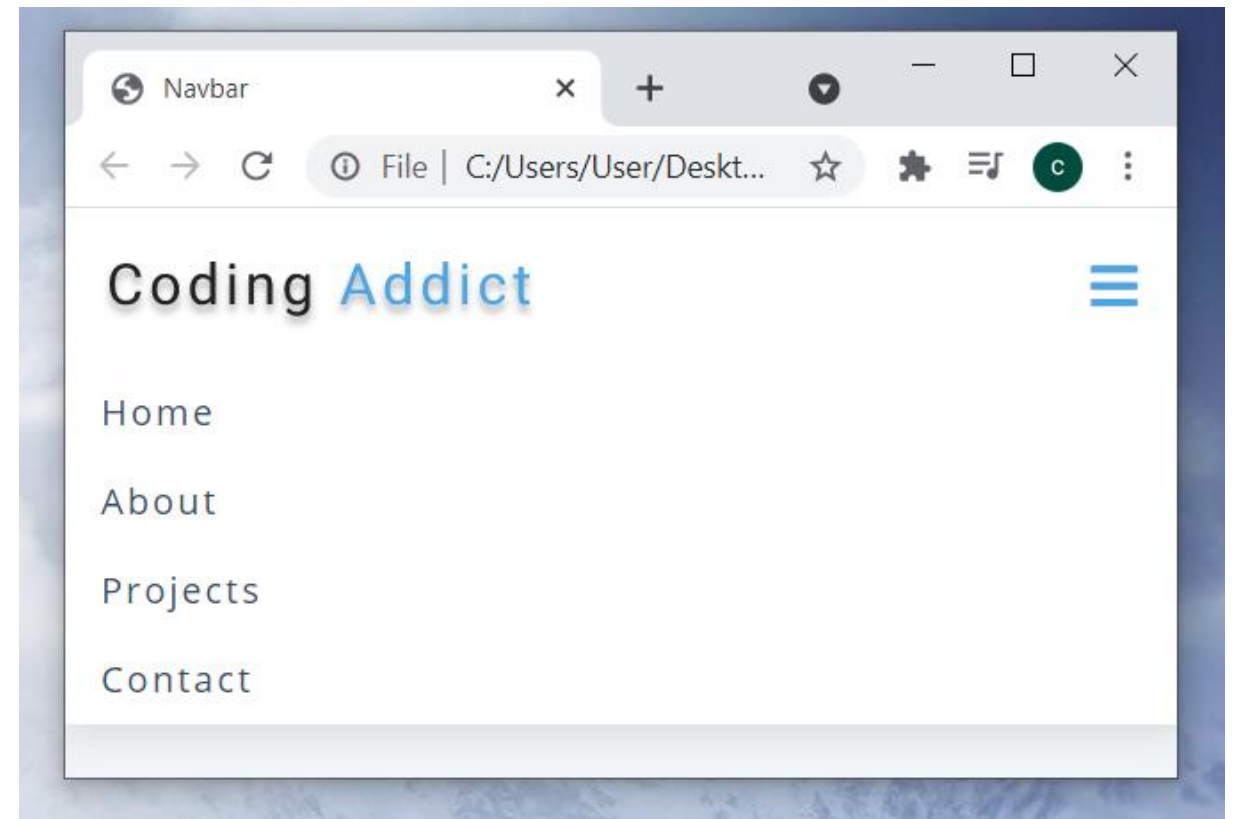
SINGAPORE UNIVERSITY OF  
TECHNOLOGY AND DESIGN

# Outline (Day 1-2, Session 2)

- Motivation and basics of Express.js
- Middleware
- Http request methods
- Routers

# Browser side: a beautiful app

- First, let's have a look at John Smilga's app:  
<https://github.com/john-smilga/node-express-course/tree/main/02-express-tutorial/navbar-app>
- Download the codes on your desktop in a folder called navbar-app.
- In your browser, this app should work.



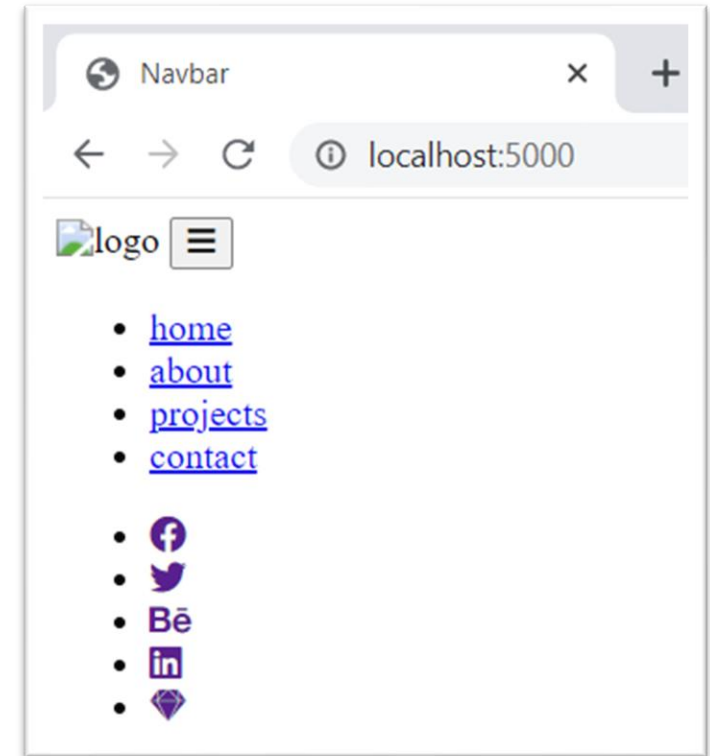
# Server side: a lame app...








- Now, let's see what happens when we serve it on a server, similar to our previous example.
- Same code than before, up to:

```
const homepage = readFileSync('../navbar-app/index.html')
```

- It does not seem to work very well.
- Why?

```
✗ GET http://localhost:5000/styles.css net::ERR_ABORTED 404 (Not Found)
✗ GET http://localhost:5000/browser-app.js net::ERR_ABORTED 404 (Not Found)
✗ GET http://localhost:5000/logo.svg 404 (Not Found)
```



Name	Status	Type
 localhost	200	document
 all.min.css	200	stylesheet
 styles.css	404	stylesheet
 browser-app.js	404	script
 logo.svg	404	text/html
 fa-solid-900.woff2	200	font
 fa-brands-400.woff2	200	font

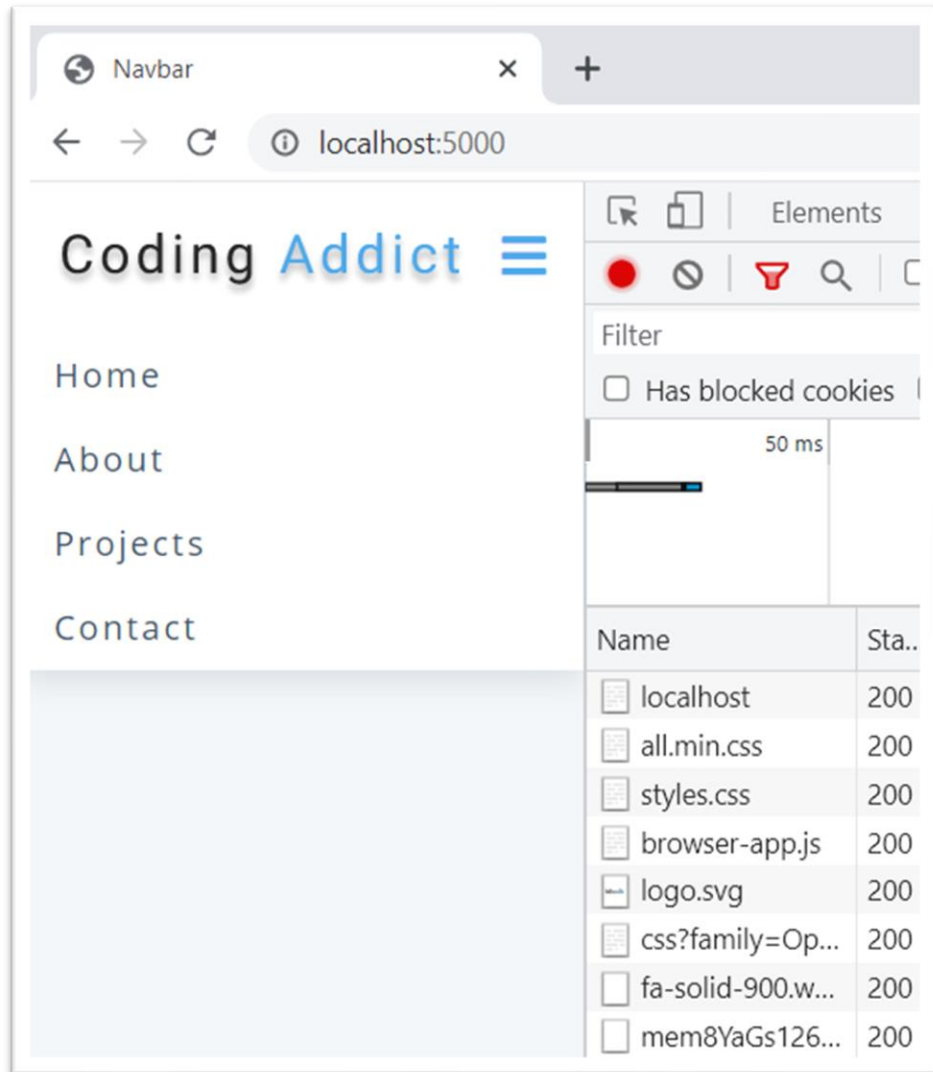
# Server side: a lame app...

- Why?
- Because we are sending back the initial html content. The browser reads the content and, every time the browser reads a path (e.g. ./styles.css), requests it from the server.
- But server side, we are only serving something if the path is localhost:5000/, or /about.
- For any other path requests, we are serving an error.
- We need to manually address the requests to all the files mentioned in index.html server-side.

```
const homePage = readFileSync('../navbar-app/index.html')
const homeStyles = readFileSync('../navbar-app/styles.css')
const homeImage = readFileSync('../navbar-app/logo.svg')
const homeLogic = readFileSync('../navbar-app/browser-app.js')

const server = http.createServer((req,res) => {
  const url = req.url
  // homepage
  if(url=== '/'){
    res.writeHead(200, { 'content-type': 'text/html' })
    res.write(homePage)
    res.end()
  }
  // about
  else if(url=== '/about'){
    res.writeHead(200, { 'content-type': 'text/html' })
    res.write('<h1>About page</h1>')
    res.end()
  }
  // styles
  else if(url=== '/styles.css'){
    res.writeHead(200, { 'content-type': 'text/css' })
    res.write(homeStyles)
    res.end()
  }
  // logo
  else if(url=== '/logo.svg'){
    res.writeHead(200, { 'content-type': 'image/svg+xml' })
    res.write(homeImage)
    res.end()
  }
})
```

# It works! 😊

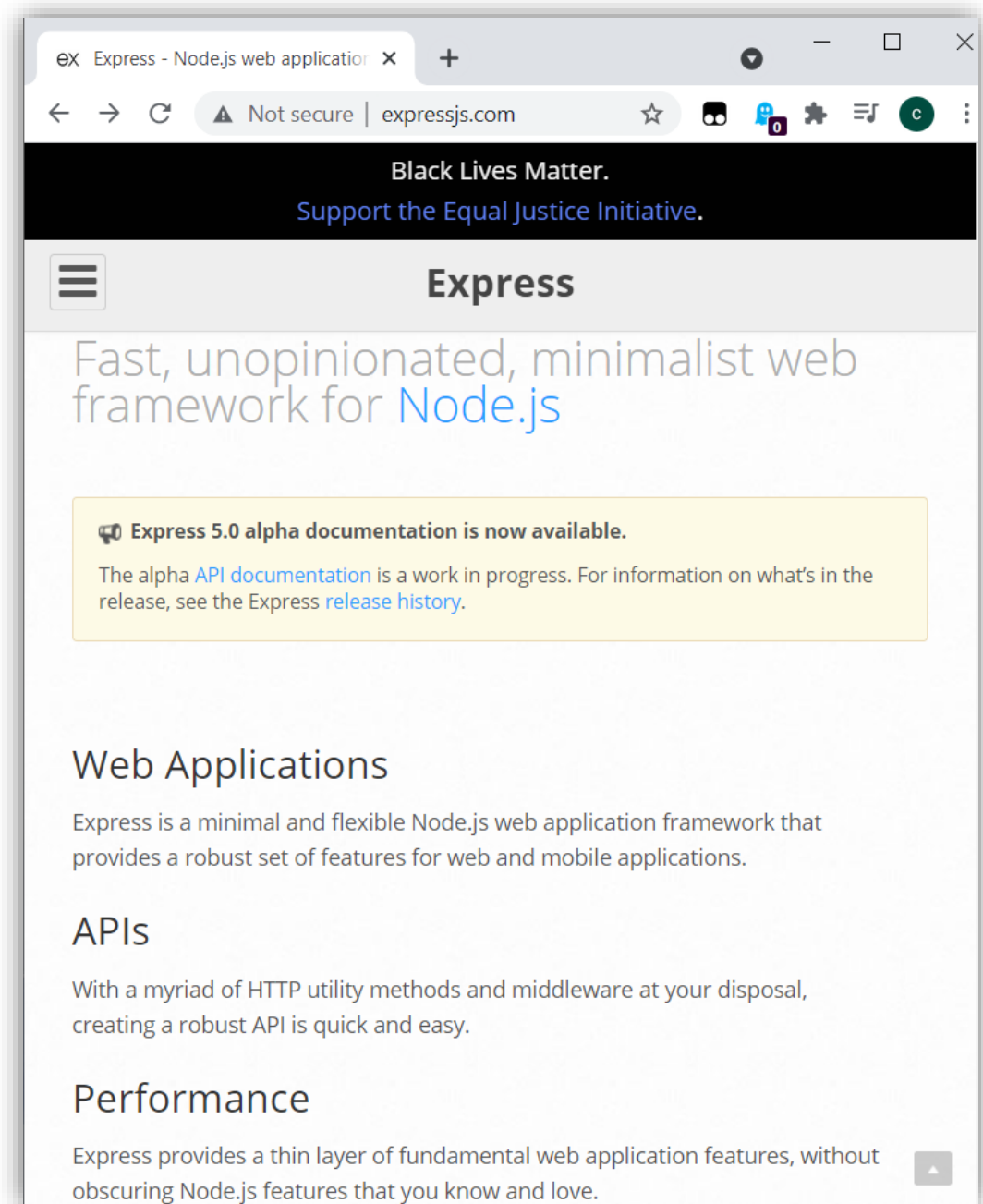


```
const homePage = readFileSync('../navbar-app/index.html')
const homeStyles = readFileSync('../navbar-app/styles.css')
const homeImage = readFileSync('../navbar-app/logo.svg')
const homeLogic = readFileSync('../navbar-app/browser-app.js')

const server = http.createServer((req, res) => {
  const url = req.url
  // homepage
  if(url=== '/'){
    res.writeHead(200, { 'content-type': 'text/html' })
    res.write(homePage)
    res.end()
  }
  // about
  else if(url=== '/about'){
    res.writeHead(200, { 'content-type': 'text/html' })
    res.write('<h1>About page</h1>')
    res.end()
  }
  // styles
  else if(url=== '/styles.css'){
    res.writeHead(200, { 'content-type': 'text/css' })
    res.write(homeStyles)
    res.end()
  }
  // logo
  else if(url=== '/logo.svg'){
    res.writeHead(200, { 'content-type': 'image/svg+xml' })
    res.write(homeImage)
    res.end()
  }
})
```

# Limitations of the http module and Express

- It becomes very messy as the app becomes more complex.
- Need to set up the path of every single resource server-side.
- These limitations prompted the use of Express, a framework for Node.js designed to ease the development of mobile, web apps and APIs.



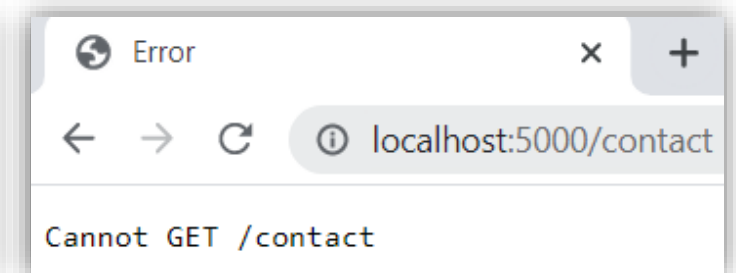
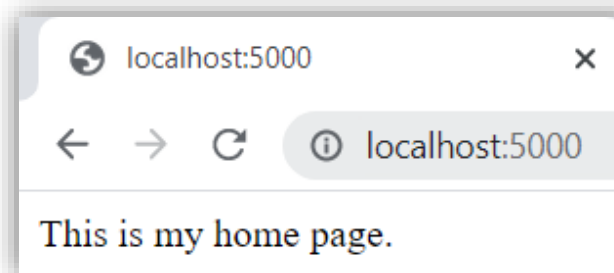
# Express: basics

- First, import and invoke the Express module with:
  - `const express = require('express');`
  - `const app = express();`
- Several methods can be used on app: **get**, **post**, **put**, **delete**, **use**, **all** and **listen**.
- The `get()` method allows you to define what happens when a user tries to get information from some route.
- Finally, the `listen()` method tells the app which port to listen and whether to do anything once it is listening.
- Note that we used **`res.send()`** instead of `res.write()` and `res.end()`. More intuitive!
- Errors are also more intuitive by default.

```
Tutorial > JS 22-appexpress.js > ...
1  const express = require('express');
2  const app = express()
3
4  app.get('/', (req, res)=>{
5    console.log('User reaches the server.')
6    res.send('This is my home page.')
7  })
8
9  app.listen(5000, ()=>{
10    console.log('Server is listening.')
11  })
12
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\22-appexpress.js
Server is listening.
User reaches the server.
User reaches the server.
User reaches the server.
█
```





# Express: basics

- The **app.all('\*', callback)** method is usually used to handle (all the) other paths.
- Note that you can send back directly html strings.
- Moreover, it is good practice to send the status code of the requests by inserting the `.status(code)` between `res` and `.send()`.

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

app.get('/', (req, res)=>{
  res.status(200).send('This is my home page.')
})

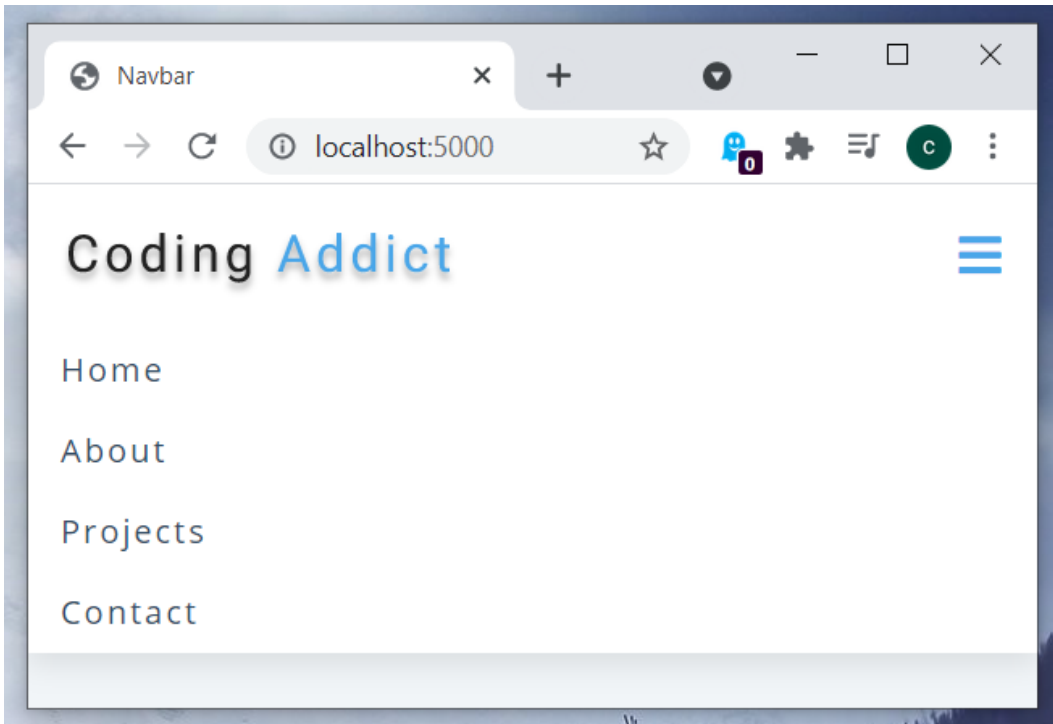
app.get('/about', (req, res)=>{
  res.status(200).send('This is my about page.')
})

app.all('*', (req, res)=>{
  res.status(404).send('<h1>Resource not found</h1>')
})

app.listen(5000, ()=>{
  console.log('Server is listening.')
})
```

# Our App example with Express

- Check out but everything should work and should be accessible.



```
Tutorial > JS 23-app2_express.js > ...
1  const express = require('express');
2  const path = require('path');
3  const app = express()
4
5  app.use(express.static('../public'))
6
7  app.get('/', (req, res)=>{
8    |   res.sendFile(path.resolve(__dirname, '../navbar-app/index.html'))
9  | })
10
11 app.all('*', (req, res)=>{
12 |   res.status(404).send('<h1>Resource not found</h1>')
13 | })
14
15 app.listen(5000, ()=>{
16 |   console.log('Server is listening.')
17 | })
18 |
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\23-app2_express.js
Server is listening.
```

# Our App example with Express:

## Explanations in short

- `path.resolve(path fragments...)` returns an absolute path formed by the path to the current folder (`__dirname`) and the relative path to `index.html`.
- Use `res.sendFile()` instead of `res.send()` since we are sending back a file.
- In a folder called `public`, we put all the static files (except `index.html`), which can be requested, meaning all the files that the server does not need to change. Here, `styles.css`, `logo.svg`, `browser-app.js`.
- `express.static(..../public)` is the set of the paths to static assets to serve.
- `app.use()` is a way to register a (group of) function(s) before executing any end route logic. More on that later.

```
Tutorial > JS 23-app2_express.js > ...
1  const express = require('express');
2  const path = require('path');
3  const app = express()
4
5  app.use(express.static('..../public'))
6
7  app.get('/', (req, res)=>{
8    |   res.sendFile(path.resolve(__dirname, '../navbar-app/index.html'))
9    | })
10
11 app.all('*', (req, res)=>{
12   |   res.status(404).send('<h1>Resource not found</h1>')
13   | })
14
15 app.listen(5000, ()=>{
16   |   console.log('Server is listening.')
17   | })
18 |
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

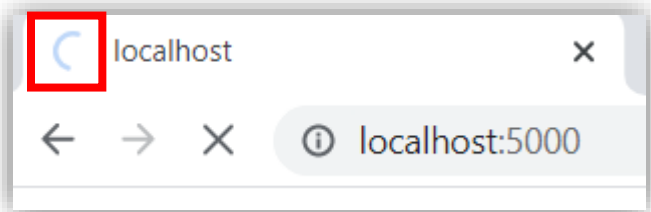
```
PS C:\Users\User\Desktop\Tutorial> node .\23-app2_express.js
Server is listening.
█
```

# Middleware: definition

- Middleware functions are functions that have access to the **req** and the **res** objects, and to the **next** middleware function in the application's request-response cycle.
- The next middleware function is commonly denoted by a variable named **next**.
- <https://expressjs.com/en/guide/using-middleware.html>

# Middleware (setup I)

- Assume that you would like to log for each request on the server the method used by the users, the url they try to access and the week day of the log.
- That would be troublesome to define it in each `app.get()` block.
- Let's create a `logger()` middleware function with inputs **req**, **res** and **next** and add it to each `get()` block.
- Works fine server-side but browser-side...



JS 24-middleware.js X

Tutorial > JS 24-middleware.js > ...

```
1  const express = require('express');
2  const app = express();
3  const logger = (req, res, next)=>{
4      const method = req.method;
5      const url = req.url;
6      const time = new Date().getDay();
7      console.log(method, url, time);
8  }
9  app.get('/', logger, (req, res)=>{
10     res.send('Home Page')
11 })
12 app.get('/about', logger, (req, res)=>{
13     res.send('About Page')
14 })
15 app.listen(5000, ()=>{
16     console.log('Server is listening port 5000.')
17 })
18
```

PROBLEMS

OUTPUT

TERMINAL

DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\24-middleware.js
Server is listening port 5000.
GET / 4

```

# Middleware (setup II)

- The middleware function must pass on to the next middleware using `next()` or terminate the cycle using `res.send()`. Now it works.
- However, this method still requires to add logger manually in each `get()` method. Moreover, logger-like functions might be quite long and complex, which makes the code less readable.
- It might be preferable to store the logger function in an external file.

```
Tutorial > JS 24-middleware.js > [🔗] logger
1  const express = require('express');
2  const app = express();
3  const logger = (req, res, next)=>{
4      const method = req.method;
5      const url = req.url;
6      const time = new Date().getDay();
7      console.log(method, url, time);
8      next()
9  }
10 app.get('/', logger, (req, res)=>{
11     res.send('Home Page')
12 })
13 app.get('/about', logger, (req, res)=>{
14     res.send('About Page')
15 })
16 app.listen(5000, ()=>{
17     console.log('Server is listening port 5000.')
18 })
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\24-middleware.js
Server is listening port 5000.
GET / 4
GET / 4
GET /about 4
```

# Middleware: app.use()

- app.use() handles this issue.
- First, create a 'logger' folder and require it in the main program.
- Add app.use(logger) and remove logger from the get() methods.
- The logger is automatically called whenever a user hits a resource.
- Caution: app.use(logger) must be placed **before** the get() methods.
- You can also input a path argument into app.use(). In that case, the middleware is only triggered if the users try to access a resource stored from this path (in the commented example, anything **after** './hobbies/').

```
Tutorial > JS 25-middleware2.js > ...
1  const express = require('express');
2  const app = express();
3  const logger = require('./logger');
4
5  app.use(logger);
6  //app.use('/hobbies', logger);
7
8  app.get('/', (req, res)=>{
9    res.send('Home Page')
10 })
11 app.get('/about', (req, res)=>{
12   res.send('About Page')
13 })
14 app.get('/hobbies/chess', (req, res)=>{
15   res.send('Chess games')
16 })
17 app.get('/hobbies/diving', (req, res)=>{
18   res.send('Dive trips')
19 })
20
21 app.listen(5000, ()=>{
22   console.log('Server is listening port 5000.')
23 })
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\25-middleware2.js
Server is listening port 5000.
GET / 4
GET /chess 4
GET /hobbies/diving 4
```

# Multiple middlewares

- It is possible to define several middlewares and to call them all in `app.use()`.
- The argument of `app.use()` must be then an array of middlewares.
- These middlewares are executed sequentially, from first to last indices.

```
Tutorial > JS logger.js > ...
1  const logger = (req, res, next)=>{
2    const method = req.method;
3    const url = req.url;
4    const time = new Date().getDay();
5    console.log(method, url, time);
6    next()
7  }
8
9  module.exports = logger
```

```
Tutorial > JS authorise.js > ...
1  const authorise = (req, res, next)=>{
2    console.log('authorise')
3    next()
4  }
5
6  module.exports = authorise
```

```
Tutorial > JS 25-middleware3.js > ...
1  const express = require('express');
2  const app = express();
3  const logger = require('./logger');
4  const authorise = require('./authorise');
5
6  app.use([logger, authorise]);
7
8  app.get('/', (req, res)=>{
9    res.send('Home Page')
10 })
11 app.get('/about', (req, res)=>{
12   res.send('About Page')
13 })
14 app.get('/hobbies/chess', (req, res)=>{
15   res.send('Chess games')
16 })
17 app.get('/hobbies/diving', (req, res)=>{
18   res.send('Dive trips')
19 })
20
21 app.listen(5000, ()=>{
22   console.log('Server is listening port 5000.')
23 })
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
Server is listening port 5000.
GET / 6
authorise
GET / 6
authorise
```



# Queries in middlewares

- Middlewares can be used to handle queries on the server.
- E.g., here, we intend to serve the hobbies pages only to our user James (in a very insecure way though...).

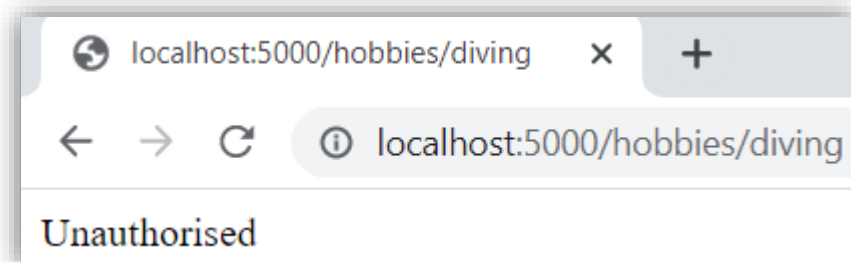
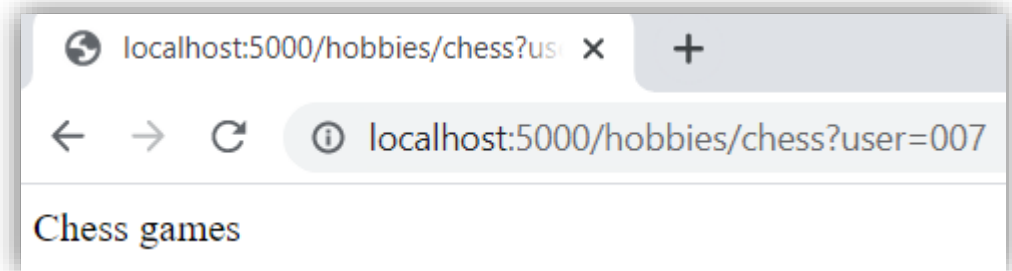
```
Tutorial > JS authorise.js > ...
1  const authorise = (req, res, next)=>{
2    const { user } = req.query
3    if (user === '007') {
4      req.user = {name: 'James', id: '007'}
5      console.log('authorised')
6      next()
7    } else {
8      res.status(401).send('Unauthorised')
9    }
10 }
11
12 module.exports = authorise
...
```

```
Tutorial > JS 25-middleware3.js > ...
1  const express = require('express');
2  const app = express();
3  const logger = require('./logger');
4  const authorise = require('./authorise');
5
6  app.use('/hobbies', [logger, authorise]);
7
8  app.get('/', (req, res)=>{
9    res.send('Home Page')
10 })
11 app.get('/about', (req, res)=>{
12   res.send('About Page')
13 })
14 app.get('/hobbies/chess', (req, res)=>{
15   console.log(req.user)
16   res.send('Chess games')
17 })
18 app.get('/hobbies/diving', (req, res)=>{
19   res.send('Dive trips')
20 })
21
22 app.listen(5000, ()=>{
23   console.log('Server is listening port 5000.')
24 })
```

# Queries (in short)

- The req.query property is an object containing a property for each query string parameter in the route.
- In the url, the query is placed at the end of the route after a question mark.
- E.g., here, we intend to serve the hobbies pages only to our user James (in a very insecure way though...).
- If a user requests this page with user = '007', the chess-game page is served. Don't forget the next() into the if block.
- Here, we also assume that all the information about user '007' are retrieved to the server from a database, e.g., as a JSON object.
- If another user requests this page with a wrong or without id, a 401 status is raised, and the middleware cycle is terminated by a res.send().

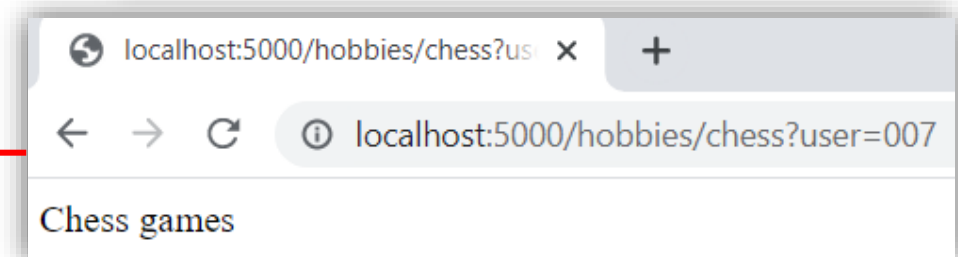
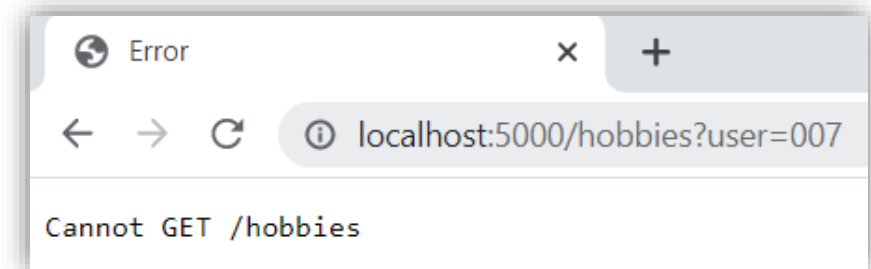
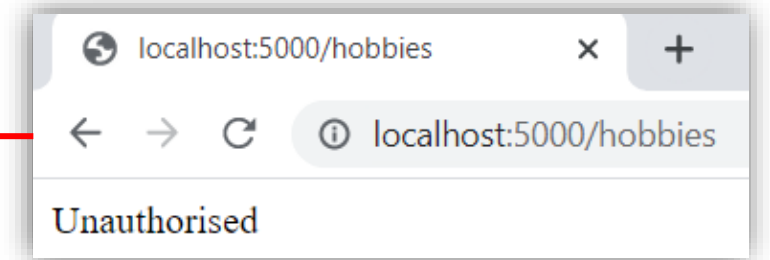
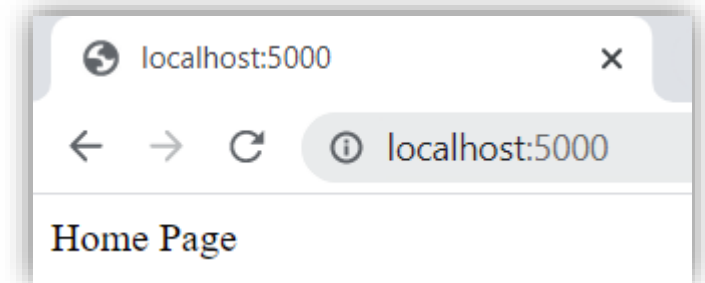
```
Tutorial > JS authorise.js > ...  
1  const authorise = (req, res, next)=>{  
2    const { user } = req.query  
3    if (user === '007') {  
4      req.user = {name: 'James', id: '007'}  
5      console.log('authorised')  
6      next()  
7    } else {  
8      res.status(401).send('Unauthorised')  
9    }  
10 }  
11 |  
12 module.exports = authorise
```



# Multiple middlewares: example

- Server-side, a request to '/' (homepage) was successful whoever the user is.
- Then, a request message to /hobbies was displayed on the console from an unknow user. Browser-side, the 'Unauthorised' message is displayed.
- A request to /hobbies from user=007 is served by a 'Cannot GET /hobbies' because this page does not exist. However, this request remains valid from user 007 and the request message is displayed on the server.
- A request to /hobbies/chess from user=007 is served browser-side.
- Note that the corresponding app.get() also displays in the server console the object **req.user** (defined in and exported by authorise.js).

```
PS C:\Users\User\Desktop\Tutorial> node .\25-middleware3.js
Server is listening port 5000.
GET / 6
GET /?user=007 6
authorised
GET /chess?user=007 6
authorised
{ name: 'James', id: '007' }
```



# HTTP request methods

- HTTP works as a request-response protocol between a client and a server.
- Defines methods to indicate the desired action to be performed on the identified resource.
- You are already familiar with the GET method which is used to request and read data sent from a source.
- POST is used to send data to a server to create/update a resource.
- PUT is used to send data to a server to create/update a resource.
- The DELETE method deletes the specified resource.

# HTTP request methods (introduction)

- HTTP works as a request-response protocol between a client and a server.
- Defines methods to indicate the desired action to be performed on the identified resource.
- You are already familiar with the GET method which is used to request and read data sent from a source.
- POST is used to send/insert data to a server to create/update a resource.
- PUT is used to send data to a server to create/update a resource.
- The DELETE method deletes the specified resource.

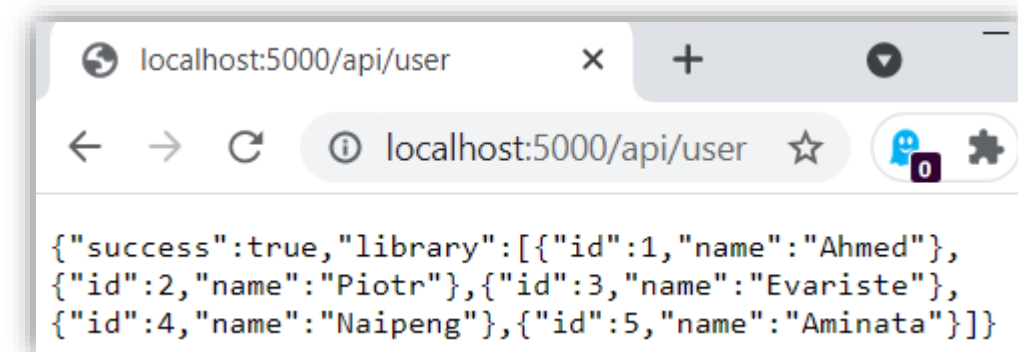
# GET method: Additional info

- You can also send back data in JSON format.
- The `res.json()` function sends a JSON response. This method sends a response (with the correct content-type) that is the parameter converted to a JSON string using the `JSON.stringify()` method.

```
Tutorial > JS 26-getmethod.js > ...
1  const express = require('express');
2  const app = express()
3  let { user } = require('./library');
4
5  app.get('/api/user', (req, res)=>{
6    res.status(200).json({success: true, library:user})
7  })
8
9  app.listen(5000, ()=>{
10    console.log('Server is listening port 5000.')
11  })
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\26-getmethod.js
Server is listening port 5000.
```



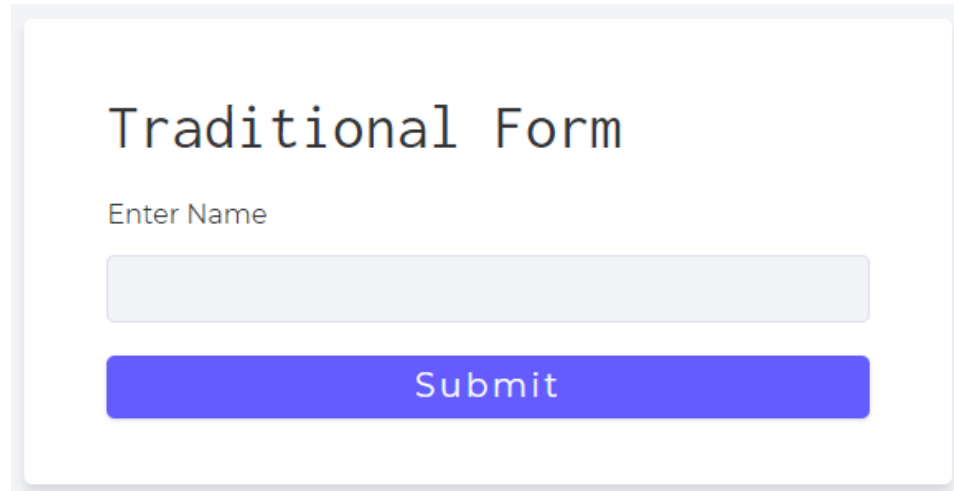
The screenshot shows a web browser window with the address bar set to `localhost:5000/api/user`. The page content displays a JSON object: `{"success":true,"library":[{"id":1,"name":"Ahmed"}, {"id":2,"name":"Piotr"}, {"id":3,"name":"Evariste"}, {"id":4,"name":"Naipeng"}, {"id":5,"name":"Aminata"}]}`. The browser interface includes back, forward, and refresh buttons, as well as a search bar and a star icon for bookmarks.

```
Tutorial > JS library.js > books
1  const books = [
2
3    {
4      id: 1,
5      title: 'Memoirs of Hadrian',
6      author: 'Marguerite Yourcenar',
7      instock: false
8    },
9
10   {
11     id: 2,
12     title: 'The Abyss',
13     author: 'Marguerite Yourcenar',
14     instock: true
15   },
16
17   {
18     id: 3,
19     title: 'To Kill a Mockingbird',
20     author: 'Harper Lee',
21     instock: true
22   },
23
24   {
25     id: 4,
26     title: 'The Girl Who Played Go',
27     author: 'Shan Sa',
28     instock: true
29   },
30
31   ]
32
33  const user = [
34    { id: 1, name: 'Ahmed' },
35    { id: 2, name: 'Piotr' },
36    { id: 3, name: 'Evariste' },
37    { id: 4, name: 'Naipeng' },
38    { id: 5, name: 'Aminata' },
39  ]
40
41  module.exports = { books, user }
```

# POST method

- Create a folder server side in the same directory than the previous js file and copy-paste the files in 'methods-public' folder at:
- <https://github.com/john-smilga/node-express-course/tree/main/02-express-tutorial/methods-public>
- Let's have a look on index.html: there is a form with 2 attributes (action and a method) some elements to style it, and a submit button sending the user to the login page.

```
<main>
  <form action="/login" method="POST">
    <h3>Traditional Form</h3>
    <div class="form-row">
      <label for="name"> enter name </label>
      <input type="text" name="name" id="name" autocomplete="false" />
    </div>
    <button type="submit" class="block">submit</button>
  </form>
</main>
```



Traditional Form

Enter Name

Submit

# POST method

- Server side, add an `app.use()` middleware to handle these public static assets.
- Also add an `app.post()` request to the login page.
- Unfortunately, if you submit a name, you will get a 404 error: Cannot POST /login

## Traditional Form

Enter Name

```
Tutorial > JS 27-postmethod.js > ...
1  const express = require('express');
2  const app = express()
3  let { user } = require('./library');
4
5  app.use(express.static('./public'))
6
7  app.get('/api/user', (req, res)=>{
8    |   res.status(200).json({success: true, library:user})
9  | })
10
11 app.post('/api/login', (req, res)=>{
12 |   res.send('POST')
13 | })
14
15 app.listen(5000, ()=>{
16 |   console.log('Server is listening port 5000.')
17 | })
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\27-postmethod.js
Server is listening port 5000.
```

Error

localhost:5000/login

Cannot POST /login



# POST method

- In order to get the form data, you must add an **express.urlencoded** middleware.
- This middleware will parse the req data in the app.post() method and will add these values in req.body
- The **extended** option is just an option to indicate which built-in library is used for parsing the data. For more details, <http://expressjs.com/en/resources/middleware/body-parser.html>

Traditional Form

Enter Name

Submit

localhost:5000/login

localhost:5000/login

POST

```
Tutorial > JS 27-postmethod.js > ...
1  const { urlencoded } = require('express');
2  const express = require('express');
3  const app = express()
4  let { user } = require('./library');
5
6  // handles the static assets
7  app.use(express.static('./public'))
8  // get the form data
9  app.use(express.urlencoded({ extended: false }))
10
11 app.get('/api/user', (req, res)=>{
12   res.status(200).json({success: true, library:user})
13 })
14
15 app.post('/login', (req, res)=>{
16   console.log(req.body)
17   res.send('POST')
18 })
19
20 app.listen(5000, ()=>{
21   console.log('Server is listening port 5000.')
22 })
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\27-postmethod.js
Server is listening port 5000.
[Object: null prototype] { name: 'Cyrille' }
```

# Routers: motivation

- Look at the example on the right. Your server will become messy very quickly...
- Moreover, many methods can be called over similar path fragments.
- In order to make the code more readable, it is common practice to first create a 'routes' folder and then to gather together all the methods to the routes with a common fragment in a router.
- In other words, the Express routers make it easy to set up any routes and respond to those routes in the proper way.

```
Tutorial > JS 29-router_1.js > ...
1  const express = require('express');
2  const app = express()
3  let { user } = require('./library');
4  // handles the static assets
5  app.use(express.static('./public'))
6  // get the form data
7  app.use(express.urlencoded({ extended: false }))
8  app.post('/login', (req, res)=>{
9    const {name} = req.body;
10   if(name){
11     return res.status(200).send(`Welcome ${name}!`)
12   }
13   res.status(401).send('No empty name please.')
14 })
15 app.get('/api/user', (req, res)=>{
16   res.status(200).json({success: true, library:user})
17 })
18 app.post('/api/user', (req, res)=>{
19   const {name} = req.body;
20   if(!name){
21     return res
22       .status(400)
23       .json({ success: false, msg: 'Please provide a name.'})
24   }
25   res.status(201).send({ success: true, person: name})
26 })
27 app.listen(5000, ()=>{
28   console.log('Server is listening port 5000.')
29 })
```

# Routers: example

- In the 'routes' folder, we created a user.js file.
- At line 2, we declare a router using `express.Router()`.
- From 6 to 18, we copied and pasted the code corresponding to the methods triggered at `/api/user`.
- Then, we replaced `app` by `router`.
- Optional: we also replaced the path `/api/user` by `/` (see why next slide).
- Finally, we export the router.

```
Tutorial > routes > JS user.js > ...
1  const express = require('express')
2  const router = express.Router()
3  let { user } = require('../library');
4
5
6  router.get('/', (req, res)=>{
7    res.status(200).json({success: true, library:user})
8  })
9
10 router.post('/', (req, res)=>{
11   const {name} = req.body;
12   if(!name){
13     return res
14       .status(400)
15       .json({ success: false, msg: 'Please provide a name.'})
16   }
17   res.status(201).send({ success: true, person: name})
18 })
19
20 module.exports = router
```

# Routers: example

- Of course, in the main file, we need to import this router (done at line 4) and we name it **user**.
- Finally, at line 12, we complete the router set up by adding the middleware `app.use('/api/user', user)`.
- Note: that's why we set up the root path at `'/'` in the previous slide. We could have done the reverse.
- Last but not least, we set up another router, `auth`, in a similar way to handle the routes to `'/login'`.
- The code on the right is now much more readable.

```
Tutorial > JS 29-router_2.js > ...
1  const express = require('express');
2  const app = express()
3
4  const user = require('./routes/user')
5  const auth = require('./routes/auth')
6
7  // handles the static assets
8  app.use(express.static('./public'))
9  // get the form data
10 app.use(express.urlencoded({ extended: false }))
11
12 app.use('/api/user', user)
13 app.use('/login', auth)
14
15 app.listen(5000, ()=>{
16   console.log('Server is listening port 5000.')
17 })
```

# Activity 2

- Set up a server as previously but with Express.js such that browser-side, at localhost:5000, if a name is submitted, the user receives a “Welcome <name>!” page.
- If an empty name is submitted, a 401 error should be triggered and the server should send back an “Please provide something” message.
- Make use of routers in order to make the code as clean and modular as possible.

# Summary

- Basics of Express
- Middleware (queries, app.use, etc.)
- Http request methods: GET and POST
- Routers