LEARN EXPRESS JS

IN 35 MINUTES.

PLATFORM :

STARTED : 17/9/22

CHANNEL :

FINISHED : 17/9/22

TUTOR :

DURATION :

GETTING STARTED

Requirement: Install node

- [ ] Open Terminal

```
> mkdir Learn Express          # Create a new folder
> cd Learn Express             # change directory into new folder
> npm init -y                  # Sets up a package.Json - where we
                                 will install all our different libraries
                                 we will be using
> Code .                       # Opens Vs Code
```

- [ ] In Vs Code, Open up the package.Json file
- [ ] Open Vs Code Terminal (Ctrl + ~) shortcut

```
> npm i express                # Installs Express Library

> npm i --save-dev nodemon     # Allows us to easily restart our server
                                 any time we make changes
```

- [ ] package.Json — complete set up

```
"scripts": {
  "devStart": "nodemon server.Js"
}
```

"devStart": "nodemon server.Js" → a file we are going to create

- [ ] server.Js

```
                               # Create this in root directory
```

Now if we run the Command npm run devStart. It is going to open our server.Js file and run all the code there

- [ ] Terminal (on Vs Code)

```
>npm run devStart
```

Now whenever we make a change in server.Js it automatically runs

Creating Express Server

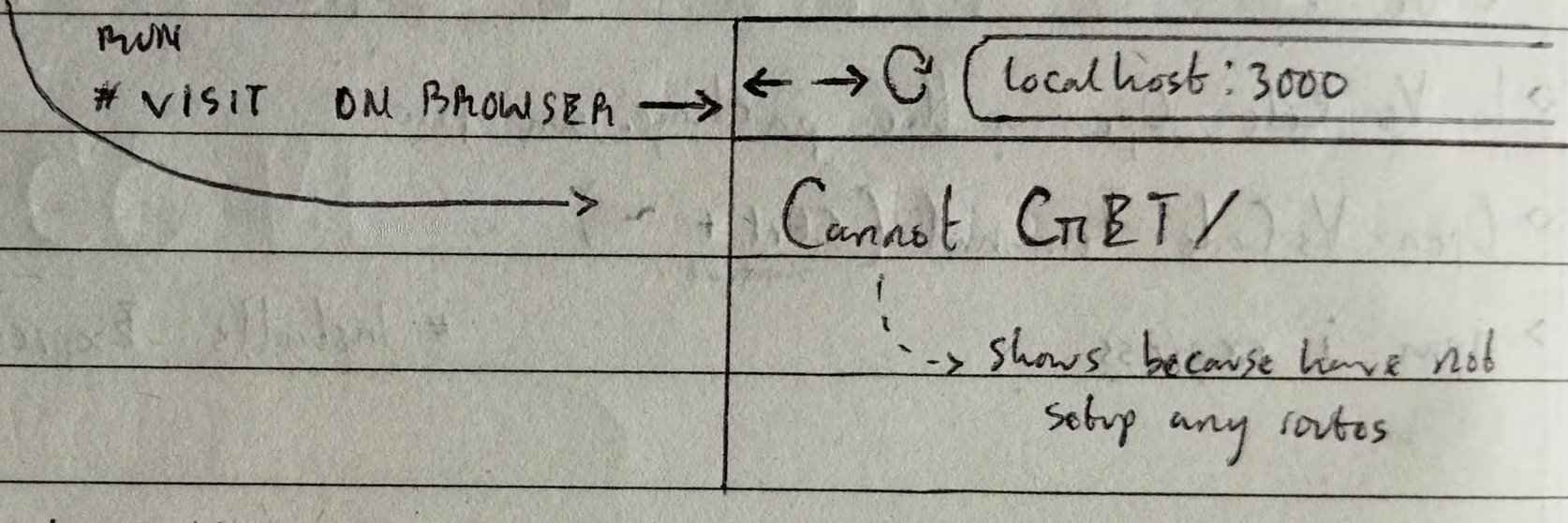
server.js

```
const express = require("express")        # We must require the express library we downloaded

const app = express()                     # By calling the express function() we create an application that allows us setup our entire server

app.listen(3000)                          # Runs Server
          Port number
```

RUN
VISIT ON BROWSER →



We can setup different routes by using app. and calling any route function

E.g app.get()
app.put()
app.delete() and others

// Confirmation ----------> # It also takes a 3rd argument "next" but we won't be using that for this tutorial

```
app.get('/', (req, res,) => {
    Console.log('Here')
    res.send("Hi")
})
```

'--> tons of other Methods

RUN →

Hi

TO RUN = Ctrl + S (restarts server)

To Send Some Status Code

```
app.get('/', (req, res) => {
    console.log("Here")
    res.sendStatus(500)
})
```

RUN →

Internal Server Error

- Usually when we send a status, we also want to send a descriptive message with it.

```
res.status(500).send("Problem Encountered")
```

RUN →

Problem Encountered

- We can also send **json** (Helpful when working with a Client API)

```
res.status(500).json({ message: "Error" })
```

RUN →

{message: "Error"}

- Sending JSON as the response

```
res.json({"message": "Successfull"})
```

RUN →

{"message": "Successfull"}

- To Send a File for the user to download

```
res.download("server.js")
```

--> Let's download this current file

RUN →

Filename [Start] [Cancel]

- To Render a File

```
res.render("index")
```

≡ We need to now create a view

> index.html

Res Methods

The 2 main res methods we will be using in the real-world are the

- render()
- json()

- [ ] views > index.html # Create folder and file

Paperlabs with the boilerplate HTML

```
...
<body>
  Hello
</body>
...
```

RUN →

Error: No default engine

... ↓

This error shows up because we don't have any view engine set up.

> The nice thing about using your server to output your views is that you can actually run your server code to generate the code inside of your views" - To do this we will be using EJS as our view engine

- [ ] Terminal (on VsCode)

```
> npm i ejs          # Installs EJS view engine
```

≡ Tell app to use EJS as it's default view engineee

- [ ] server.js

```
Const express = require("express")
Const app = express()

+ app.set('view engine', 'ejs')
                          ----> ejs now set

app.get('/', (req, res) => {
   console.log("Here")
   res.render("indexe")
})
```

app.listen(3000)

≡ Next: rename index.html to index.ejs and then install the extension called "EJS language support"

≡ Next: restart server in Terminal "> npm run devStart

Hello

≡ To Pass data from server —into→ views (add Second Parameter)

res.render("index", { text: "World" })

≡ To Access the data passed by the server in your views

○ views > index.ejs

```
<body>
  Hello <%= text %>
</body
```

RUN →

Hello World

≡ If we don't use the correct Identifier passed by the server. We get an Error

res.render("index", { text1234: "World" }) RUN →

Error

we add

or any value

≡ To Fix ^ locals.text || 'Default'

```
<body>
  Hello <%= locals.text %>
</body>
```

Hello

≡ Add a default if identifier doesn't exist

```
<body>
  Hello <%= locals.text || 'My Default' %>
```

My Default

Router

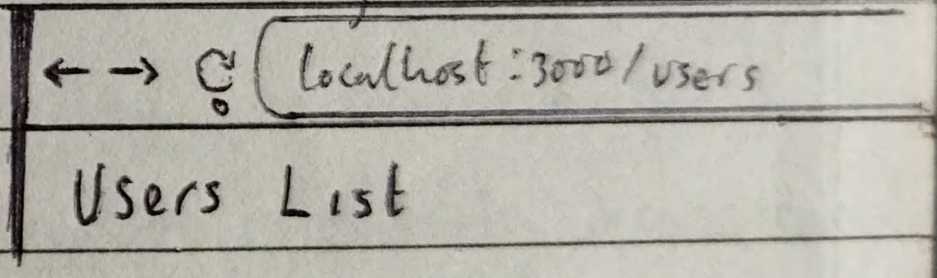
A Router is a way for us to create another instance of our application. that is it's own mini application, And then we can insert it into the main application

Creating a Router

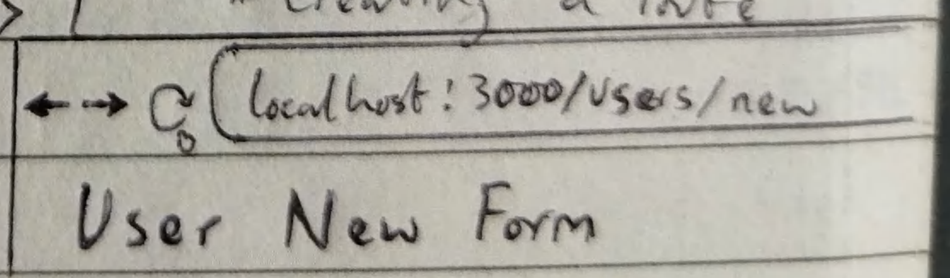
routes > users.js # Create this

```
const express = require('express')          # We must require express
const router = express.Router()             # This router now creates a mini-app
                                            that is independent from our
                                            main app in server.js

router.get("/users", (req, res) => {        # Creating a route
    res.send("Users List")
})
```




```
router.get("/users/new", (req, res) => {    # Creating a route
    res.send("User New Form")
})
```




```
module.exports = router
```

Before Inserting this router into the main application. Lets remove /users from both routes above

... "/Users" to "/"

... "/Users/new to "/new"

VERY IMPORTANT

Add our Router now inside the main application

server.js

```
3)
const userRouter = require("./routes/users")
```

```
app.use("/users", userRouter)
```

You can do this for each mini-app you want in your application.
And it will just make your code super clean and much better

Dynamic Routes

routes > users.js

```
... // To Create new user (We send a POST request)
+ router.post('/', (req, res) => {
+     res.send('Create User')
+ })

// To use dynamic URL to access any user in our application (GET)
+ router.get('/:id', (req, res) => {
+     // To access:
+     req.params.id
+     res.send(`Get User With Id ${req.params.id}`)
+ })
```

localhost:12

Get User With id 2

NOTE: Your dynamic url routes must always be located at the bottom of your routes as it should in order to avoid not working (due to matching incorrections)

```
// To Update an existing user (We send a PUT request)
+ router.put("/:id", (req, res) => {
+     res.send(`Update User With ID ${req.params.id}`)
+ })
```

localhost:51

Update User With ID 51

```
// To delete an existing user (We send a DELETE request)
+ router.delete("/:id", (req, res) => {
+     res.send(`Delete User With ID ${req.params.id}`)
+ })
```

localhost:166

Delete User With ID 166

1 Routes & Different Requests

NESTED ROUTES

If we have more than 1 type of request (Get, Put, Post, delete e.t.c) repeating the exact same route (in our case "/:id" were repeated for 3 different requests)

Express provides us with a way to simplify that

routes > users.js ☐ Full Code

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

router.get("/", (req, res) => {
    res.Send("Users List")
})

router.get("/new", (req, res) => {
    res.Send("User New Form")
})

router.post("/", (req, res) => {
    res.Send("Create User");
})
```

```
+ router
+   .route("/:id")
+   .get((req, res) => {
        res.send(`Grab User With ID ${req.params.id}`)
    })
+   .put((req, res) => {
        res.send(`Update User With ID ${req.params.id}`)
    })
+   .delete((req, res) => {
        res.send(`Delete User With ID ${req.params.id}`)
    })

+ module.exports = router
```

□ End of Code

And Just Like That

We have organized all those requests that share the exact same route into what ~~was~~ we see above.
^

ROUTER.PARAM

We have access to a param function in our router

```
routes > users.js
})

router.param("id", (req, res, next, id) => {
    console.log(id)
})

module.exports = router
```

Explanation

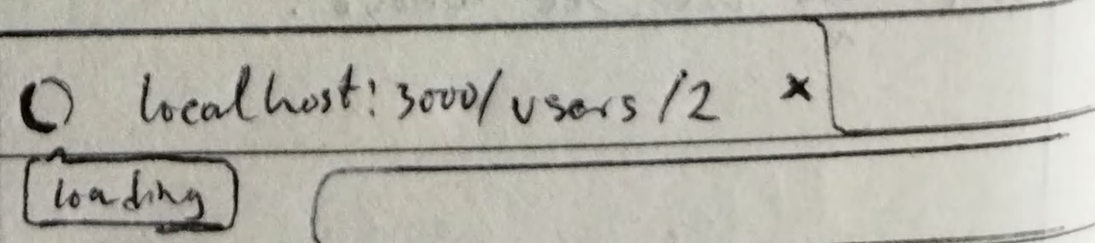
What this router.param does is that anytime it finds any routes parameter that matches the first argument you pass into it.

It is going to run

NOTE: In our case our above get, put and delete routes all have "id" param so this is going to run for the 1st, 2nd or 3rd if we decide to visit any of the routes

Once you RUN the Code and REFRESH Your browser.

It LOADS Infinitely



MIDDLEWARE

Why does this happen?

This happens because when our router.param finds the first route with "id" for it's parameter. * It Jumps and executes router.param(), We must use the next() inside router.param if not execution freezes there

up of our code

```
router
  .route("/:id")
→.get((req, res) => {        # Pauses Here → Then It Jumps
    ...                                          ↓
router.param("id", (req, res, next, id) => {     # Executes this
                                                 ↓
  console.log(id)                                # Doesn't see a next()
})                                               so it freezes here
```

type of

router.param is a Middleware.

What is a Middleware in Express?

It is code that runs between a request that is being sent to your server and the actual response being returned to the user

a

In our case of using Middleware in router.param we have to use the next(), to run the next which is basically Jumping back to our paused code and continuing execution.

routes > users.js

```
router.param(":id", (req, res, next, id) => {
  console.log(id)
  next()                    # we shouldn't freeze here any longer
})
```

accessible to all matching

Using the router.param middleware to get data from one single pla

```
router.get ((req, res) => {
    console.log (req.user)
    ...
})
...↓

const users = [{ name: "kyle" }, { name: "sally" }]
router.param ("id", (req, res, next, id) => {
    req.user = users [id]
    next()
})
```

CONSOLE

My { name: "kyle" }

if you visit localhost:3000/users/1

CREATING A MIDDLEWARE

0 Server.js

...↓

app.set("view engine", "ejs")

+5 app.use(logger) # middleware

..↓

+1 function logger(req, res, next) {

+2 console.log(req.originalUrl)

+3 next()

+4 } =>

CONSOLE		RESULT
/users	# route you visit	Any Page we visit on our app.
/	# route you visit	We should now see the route in
/users/2	# route your visit	our CONSOLE

app.listen(3000)

≡ We can make our custom logger middleware exclusive to a function

- app.use(logger) # - means remove line

..↓

U app.get("/", logger, (req, res) => { # U means update like

...

})

CONSOLE		RESULT
/	# route works	Our logger middleware will now
	# route doesn't	only run for this route
/	# route works	

Static Files & Middlewares

Theres quite a Few built-in middlewares in Express and one of the most common ones, we are going to be using is for serving out Static Files.

Things Like:

- Static HTML, - Static CSS, - Static Javascript

Change our index.ejs into a static HTML file

views > index.ejs rename to index.html

```
<body>
    Hello World
</body>
```

Update like

Move index.html into a newly created folder public

public

Public is the standard naming Convention for static Folders

<> index.html

moved

server.js

Remove our "index" route renderer

```
- app.get('/', (req, res) => {
-     console.log("Here");
-     res.render('index');
- })
```

≡ To Serve our static file

+ app.use(express.static("public")) # This will now serve all the static files from our public folder. ALL

	localhost:3000
≡ To Prove ALL our static files are served inside our public folder. Let's create another static file	Hello World

☐ public

+ ∨ test # Create folder

+ <> ff.html # Create file

☐ public > test > ff.html

... * populate with Boilerplate HTML

<body>

asdfasd

</body>

Now if we visit this path, it should be able to Access Our static file

localhost:3000/test/ff.html
asdfasd

CONCLUSION

We see that this is a super handy way to build and render out some static HTML, or CSS or Javascript

Forms & Middlewares

Another useful Instance where we would use some Middleware, is for parsing all the information sent to your server from forms or Json requests

Let's create a new ejs file that contains a form

views *

v users # Create folder

<> new.ejs # Create file

views > users > new.ejs

```
...
<body>
  <form action="/users" method="POST">
    <label> First Name
      <input type="text" name="firstName" value="<%= locals.
                                                  firstName %>" />
    </label>
    <button type="submit"> Submit </button>
  </form>
</body>
```

routes > users.js

Let's Render our new form file

```
router.get("/new", (req, res) => {
    res.render("users/new")
})
```

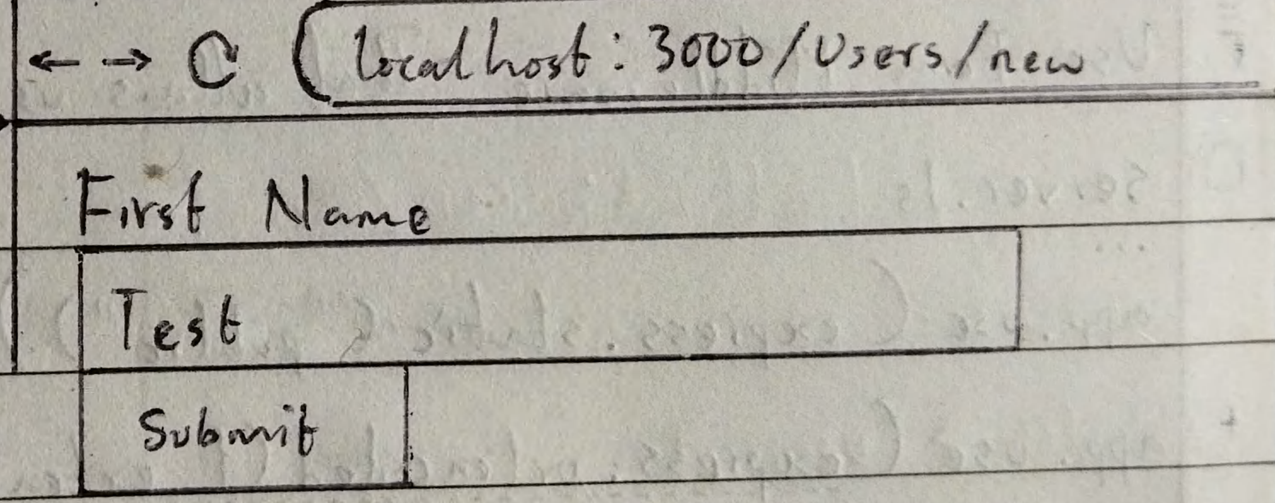
Notice: we already have already have a post route setup for when form is submitted

```
router.post("/", (req, res) => {          # matches forms action="/users"
    res.send("Create User")
}
```

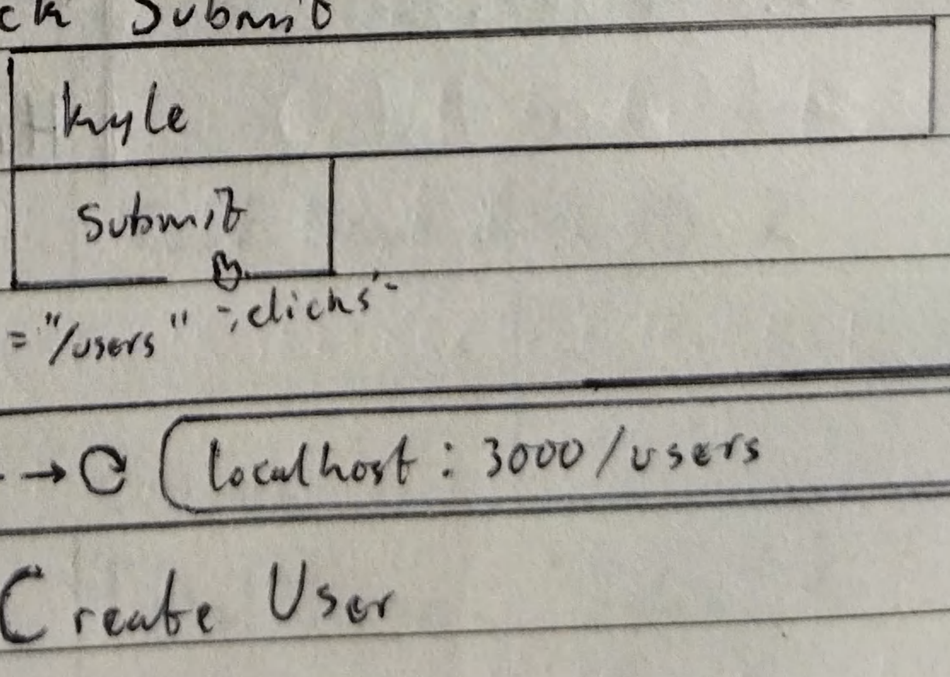
Add Placeholder FirstName

```
router.get("/new", (req, res) => {
    res.render("users/new", { firstName: "Test" })
})
```

VISIT LOCATION →



Enter Some Value and Click Submit



Sadly:

Right now we are not accessing the information submitted

views > users.js

To retrieve the Information Submitted

```
router.post("/", (req, res) => {
    console.log(req.body.firstName)   # data from <input ... name="firstN
    res.send("Hi")
})
```

Enter kyle again and Submit

← → C	localhost:3000/users/new
Error: cannot read property of undefi	

WHY AN ERROR?

This is because by default Express does not allow us access the body, except we use a middleware

Use Form Middleware - That allows us access information coming from Forms

server.js

```
...
app.use(express.static("public"))
app.use(express.urlencoded({ extended: true }))
```

Enter kyle and then Submit again

← → C	localhost:3000/users/new
Hi	

Making Things Interesting

routes > users.js

```
router.post("/", (req, res) => {
+   const isValid = true
+   if (isValid === true) {
+       users.push({ firstName : req.body.firstName })
+   res.redirect(`/users/${users.length - 1}`)
+   } else {
+       console.log("Error")
+       res.render("users/new", { firstName: req.body.firstName })
+   }
• })
```

```
... ↓
router
  .route("/:id")
  .get((req, res) => {
+     console.log(req.user)
    ...
  })
```

EXTRAS (Really Important)

In order to be able access Json, we need to use a Middleware provided to us by Express

- server.js

```
app.use(express.urlencoded(...))
```

+ `app.use(express.json())` * Allows us parse Json information from the body. Similar to urlencoded for form information

≡ How Do We Access Query Parameters?

localhost:3000/users?name=kyle

- routes > users.js

```
router.get("/", (req, res) => {
    console.log(req.query.name)
    res.send("Users List")
})
```

← → ↻ localhost:3000/users?name=kyle
User List

CONSOLE

Kyle

DONE