Jan Feb

Express JS

Of

is a web framework used to build server side applications, so we can build APIs for clients & that API can perform server side tasks like saving data to a database. Can build real time applications.

→ Need Node
→ npm
→ Postman.

→ mkdir express JS
→ cd express JS

→ npm init -y.
→ npm i express

— we can see express installed in our package.json.

→ npm i -D nodemon // installing nodemon locally.

we can see it in node_modules/bin.

→ /src/index.js

→ set up a script in package.

"start": "node ./src/index.js"

↑
referring to node command.

& "start:dev": "nodemon ./src/index.js"

↑
this will restart our server auto upon changes.

→ npm start.
→ npm ~~start~~ run start:dev.

→ index.js

const express = require('express') // imports express library

- ~~test~~ const app = express();

Creates an instance of express application. This alone wouldn't be able to do anything. We need to setup a port to listen to requests coming in.

- const PORT = 3001;
app.listen(PORT, () => console.log(`....`));

↑ A callback function.

- when we visit localhost:3001

Cannot get /.

Cuz we haven't set up any routes yet.

02

index.js

GET Req.

```
app.get('/groceries', (req, res) => {
    res.send([ {
        id: ?,
        name: ?
    } ]);
});
```

- Suppose we're making a server that retrieves grocery items
- In our get request, we pass in two params, first is path (/groceries), second is a callback func.

↓

It has 2 args, req → gives all info about the client i.e. cookies, IP Address, headers

& res → handling sending response back to the client.

~~Get~~ const groceryList.

03 Post

app.post('/groceries', (req, res) => {
  console.log(req.body);
  response.send(201);
});

body is a prop that has all the properties that are inside the req body.

head to postman →

- New → HTTP Req → ~~Get~~ POST → localhost:3001/groceries

[SEND]

↑ Created 201 response.

~~At~~ terminal → we see undefined since we didn't create a body.

- Postman → Body → Raw & JSON →

{
  "item":
  "quantity":
}

- SEND
- we still see undefined.

- problem is we're sending a req body to the server but the server does not know how to parse the data being sent. enable.
- we need to ~~implement~~ MIDDLEWARE from. for it to parse correctly.

- Body → x-www-form-urlencoded
  { standard way that form actually serialize data when it comes to POST requests. }
  → we still get undefined

Middleware – a function that needs to be implemented right in the middle of 2 main functionalities.

Suppose we want to retrieve the grocery list
So in between the user requesting to retrieve the grocerylist & us sending a response, we can implement middleware that'll perform a task like (login). Implemented after we create app instance.
→ app.use(express.json()); → console logging

→ Now send a post req.
we see a json response in our terminal.

- express deprecated res.send(201)
  using res.sendStatus(201);

adding → app.use(express.urlencoded());

- app.post(
  groceryList.push(request.body).
  ↳ updating our directory.
  );

→ after sending a POST req in postman, send a GET Request & we can see our updated directory grocery list.

04 Middleware

- Middleware is a function implemented during the lifetime of our req-res cycle.

- In express, we can think every other parameter after the route name as middleware.

- app.get('/groceries', (req, res, next) => {

});

next parameter is a function that allows us to invoke the next middleware.

- Thus every middleware has 3 functions
  - → req
  - → res
  - → next.

- Implement a middleware in app.get

we do not get a response back

That's because we didn't send anything back to the client in this middleware.

So we add next(); in our middleware.

- Implementing this middleware above us →

```
app.use((req, res, next) => {
    console.log(req.url);
    next();
});
```

URL parsing

we can also use

console.log(`${req.method}: ${req.url}`);

— deprecated, undefined, extended: provide extended options

changing to app.use(express.urlencoded({extended: true}));

— middleware registration has to be in right order.

Route Parameters

what if we wanted to get a specific grocery from the grocery list. We'd need to route parameters.

— We'll extend our grocery endpoint.

app.get('/groceries/:item', (req, res) => {

});

const {item} = req.params

const groceryItem = groceryList.find((g) => g.item === item)

predicate

res.send(groceryItem);

we'll also need to handle status codes when no grocery item has been found.

06 Routing

When it comes to larger applications we'd want to split our application using routing.

src/routes/groceries.js

```
const { Router } = require('express')
const router = Router();
```

↓ Creating an instance of Express list.

Importing Router function from Ex

- Copying get, post methods from index to routes/groceries.

```
module.exports = router; //exports our router
```

index.. - `const groceriesRoute = require('./routes/groceries');`

//importing our route.

```
app.use(groceriesRoute); // register the route.
```

↓

```
app.use('/api', groceriesRoute);
```

[Prefixing - what most companies do.

```
app.use('/api/groceries', groceriesRoute);
```

Allows us to remove, ~~route~~ path from our groceries.js

```
like router.get('/', (req, res) => {
});
```

routes/market.js // Another endpoint.

Or

Query Parameters — Adding KMs in supermarket

— Suppose we want to filter out markets that are less than 2 km away.

— we can use a.. query market.

market.js — console.log(req.query)

Postman — :3000/api/markets?km=3&sortBy=ASC.

we can see this in console.

market.js — router.get("", (req, res) => {

const {km} = req.query;

const parsedKM = parseInt(km);

if (!isNaN(parsedKM)) {

Note Number.

const filter = supermarkets.filter(

(s) => s.km <= parsedKM

res.send(filter);

} else {

res.send(supermarkets);

08. Cookies

Blocks of data that are sent from the server (express API) to the client (web browser) / to them.

Need –
HTTP is stateless. No information is shared. Upto 2 subsequent HTTP calls.

So we use cookies which is sent by the server & stored on the client side & when the client makes another req. the cookie is sent too & the server can check if the cookie is valid & it can show some info about the client.

groceries/routes.js → res.cookie('visited', true, {
maxAge: 10000,
});

– Downloading npm cookie-parser. It's a middleware.
– Importing cookie parser in index.js
– We need to call this middleware by app.use

groceries.js – ~~for~~ → console.log(req.cookies);
→ get(/:item)

– We can build more persistent & stateful servers now.

09. Sessions

Saving info on client-side using cookies is not safe. So we use sessions to store info on our server.

npm i express-session

index.js const session = require('express-session')
app.use(session());

passing in certain values secret
resave
saveUninitiated.

groceries router.get('/cart')
router.post('/cart/item')

if → res.send(req.session?)

it's only an object if we write (req.sessionID) & send POST req, we'll see diff. IDs returned everytime coz we have not modified the session. So this session is going to be unique everytime unless we modify it. Once we modify our session, it means that the session ID corresponding to that session object that was modified is going to be relevant to every single subsequent requests that is being made with the same session ID.

// modifying session

```
const {cart} = req.session;
if (cart) {
    ~~const {items} = cart;~~
    ~~items.push(cartItem);~~
    req.session.cart.items.push(cartItem);
} else {
    req.session.cart = {
        items: [cartItem],
    }
```

If the cart doesn't exist we'll create an object & we'll add the items (a property) to the cart object. & we'll assign this to an array that has the cart items there already.

→ res.send(cart);

Send a post req on cart/item, you can see a cookie has been created.

Now that we have a cookie on the browsers or client side, we can make additional req to our API that'll send us cookie, the server is going to handle the cookie by associating this cookie to corresponding session ID.

router.get('/cart', (req, res) → {
  const { cart } = req.session;
  if (!cart) {
    res.send('You have no items in your cart');
  } else {
    res.send(cart);
  }

Postman

GET ~~localhost~~:/api/groceries/~~ps~~shopping/cart.
→ you got no cart!

POST /cart/item.

GET. /cart You see your cart.

localhost → you got no cart!

Coz we don't have any cookies stored there, if we manually add the same cookie on postman we can see this cart there.

This cookies is stored on the Server Side.

10 — /routes/auth.js

Auth

— Create a middleware that's gonna check if user is logged in
inside ~~index.js~~ groceries & market.js

We need to setup a database ← If we restart our server all sessions will be lost from the memory

4.

MongoDB

— Install mongodb. (website)
— Install mongoose (npm i mongoose)
↙
JS library that allows linking and interacting with a database.

— NoSQL database
— good for beginners.

— Has also an ORM
(Object Relational Mapper)
↓
= Map objects into the database tables
— They abstract all of the raw data that we'll have to write so we don't have to learn actual SQL.

~~database~~ server/database.

/index →

const mongoose = require('mongoose') Port No.

mongoose.connect('mongodb://localhost:27017/

expressJS)

. then

. catch 127.0.0.1:

else it'll throw error

It defines what our data should look like

src/database/schemas/User.js

Under auth.js → router.post('/register')

— Asynchronous programming is a technique that enables to start a potentially long running task & still be responsive to other events while that program runs, rather than having to wait.

— Promises are used to handle async ops in JS.

— send a POST Req
— to go to Mongo Compass
— See the user database ! Tada !!

Hashing Passwords

npm i bcryptjs

with encrypting, we can recover any text with hashing we cant.

src/utils/helpers.js → export to auth

hash before creating a new user.

helpers.js → compare Password

Modify auth.js login route

13
Username & password auth

- Dropping users collection in compass
- setting email as unique (user.js) & deleting username field.
- changing auth.js login route & modifying session in it
- still if we stop the server all our session data will be gone, we want to store our data not in memory but in a database (session Store) later on.

14
PassportJS

→ A middleware library that takes care of auth process for us - It also serializes the user into the session so we dont have to manually update the session.

- npm i passport passport-local
  strategy

- import in index.js & app.use it

15
Passport Local Strategy

src/strategies/local.js

add login method in auth.js

Send a post req, we'll get an error that we're not serializing our user.

13th Feb.

16 Serialize/ Deserialize

local.js → serializeUser
deserializeUser.

Deserialize. groups.js → req.session.user / req.user.

17 sessionStore

index.js → working of memoryStore → Xdisadv.

→ npm i connect-mongo → index.js (use mongoStore).

restart server
& send a get req
we're still logged in!

18. OAuth2

→ discord/developers/applications → new app.

Create a new redirect
callback /api/auth/discord/redirect.

→ $ npm i passport-discord

strategies → discord.js

register on index.js

comment out local.js

schemas → DiscordUser.js

19 Jest & unit testing

→ $ yarn add -D jest.

$ npx jest --init.

src/__tests__

src/controllers/auth.js

src/__tests__/controllers/auth.test.js

"Stubs"

"Assertions" "Mocking".

20 Unit Testing Discord strategy Verify function.

/strategy/discord.js → Make a named function discordVerifyFunction. for testing

/tests/strategies/discord.spec.js