**3. Understanding the Basics**

26. Creating a Node Server

**NodeJS**-ის გამოყენებისას, უნდა გავითვალისწინოთ, რომ მასში არსებობს ყველაზე ხშირად გამოყენებადი, ანუ ძრავის მოდულები:

**http, https, fs, path, os.**

აქედან სერვერის მხარის გასამართად, ყველაზე გამოსადეგი მოდულებია **Http**და **Https**მოდულები.

თუ **require()** მეთოდში ჩავსვამთ ისეთ სტრინგს, რომელიც ნოუდის ქორში იმყოფება, იგივე სახელით ფაილიც რომ გვქონდეს, პრიორიტეტს მოდულს მისცემს, ამიტომაცაა საჭირო **./**სინტაქსის გამოყენება.

შემდეგ, ვიწყებთ **http**მეთოდის  გამოყენებას, რომელსაც ცვლად **const http**-ში ვინახავთ. აღნიშნულ ძრავის მოდულს გააჩნია თავისი მეთოდები და ფროფერთიები, ყველაზე მნიშვნელოვანი კი არის, ე.წ. **createServer()** მეთოდი, რომელიც არგუმენტად იღებს ქოლბაქ ფუნქციას, რომელსაც აქვს 2 არგუმენტი - **req**და **res**, რომლებიც, შესაბამისად, გარკვეულ დეითას აბრუნებენ.

მასში ამ ქოლბაქ ფუნქციის ჩასმა რამდენიმენაირად შეგვიძლია, ერთ-ერთია **arrow function**-ის გამოყენება, ანუ შიგნითვე რომ ჩავაშენოთ.

**3. Understanding the Basics**

26. Creating a Node Server

მას შემდეგ, რაც ფუნქციას ჩავაშენებთ, აღნიშნული **http**დააბრუნებს ე.წ. **server**-ს, რომელზე დასაკვირვებლადაც სასურველია ცვლადში შევინახოთ ეს გამოძახებაც.

შემდეგ, ამ ცვლადს (**server**) თუ ობიექტის სტილში მივწვდებით, ანუ დოტ ოპერატორით, ვნახავთ, რომ რამდენიმე მეთოდი აქვს მიბმული, აქედან ყველაზე მნიშვნელოვანია **.listen()** მეთოდი. იგი არგუმენტად იღებს პორტის რიცხვს. თუ ცარიელს დავტოვებთ (რამდენადაც იგი ოფშენალია), იგი **http**-ის შემთხვევაში მე-80 პორტზე დაიწყებს მოსმენას.

მას შემდეგ შეგვიძლია უბრალოდ ბრაუზერიდან გამოვიძახოთ აღნიშნული პორტი, რომელიც **localhost**-ზე იქნება მიბმული და თუ**http.createServer()**-ფუნქციის ქოლბაქის **req**არგუმენტს დავაქონსოლლოგებთ, ვნახავთ, რომ შემომავალი რექვესტი გვაქვს, რომელსაც **.listen()** მეთოდი უსმენს.

*კოდი შემდეგში...*

**3. Understanding the Basics**

26. Creating a Node Server

**app.js:**

1. const http = require('http');
3. const server = http.createServer((req, res) => {
4. console.log(req)
5. })
7. server.listen(3000)

**3. Understanding the Basics**

27. The Node Lifecycle & Event Loop

რამდენადაც **NoseJS**ივენთ ლუპს იყენებს, რათა მუდმივად დააკვირდეს ცვლილებებს, მისი შეწყვეტა მარტივი ბრძანებით შეგვიძლია, ჩვენს მაგალითში რომ ჩავსვათ:

1. const http = require('http');
3. const server = http.createServer((req, res) => {
4. console.log(req)
5. process.exit()
6. })
8. server.listen(3000)

**3. Understanding the Basics**

29. Understanding Requests

რექვესტების ჰენდლინგისას, მნიშვნელოვან დეითას წარმოადგენს **url, method** და **headers**, ისინი ჩვენს კოდში:

1. const http = require('http');
3. const server = http.createServer((req, res) => {
4. console.log(req.url, req.method, req.headers)
5. })
7. server.listen(3000)

**3. Understanding the Basics**

30. Sending Responses

რესპონსების ჰენდლინგისთვის მრავალი გზა არსებობს. მათ შორისაა **.setHeaders()**მეთოდი, რომელიც პირველ არგუმენტად ჰედერის სახელს, ხოლო მეორედ მნიშვნელობას იღებს.

ასევე შეგვიძლია ჩანკებად გავუგზავნოთ **html**ფაილი,**.write()** მეთოდის გამოყენებით.

მნიშვნელოვანია ვიცოდეთ, რომ **.end()** მეთოდის შემდეგ აღარაფერი აღარ უნდა დავწეროთ, რადგან იგი დასრულებას ნიშნავს და დაერორდება.

**app.js**

1. const http = require('http');
3. const server = http.createServer((req, res) => {
4. console.log(req.url, req.method, req.headers)
5. // process.exit()
6. res.setHeader('Content-Type', 'text/html');
7. res.write('<html>');
8. res.write('<head><title>My first page</title></head>');
9. res.write('<body><h1>Hello from my Node.js server!</h1></body>');
10. res.write('</html>');
11. res.end()
13. })
15. server.listen(3000)

**3. Understanding the Basics**

32. Routing Requests

**ჰედერების სოურსი:**

*https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers*

**3. Understanding the Basics**

32. Routing Requests

ჩვენ შეგვიძლია შიგნითვე გავაკონტროლოთ როუტინგი, მაგალითად ქონდიშენალ სტეიტმენტში მოვუსმინოთ **req.url**-ს და მისი მიხედვით მოვახდინოთ კოდის ეგზეკუშენი:

1. const http = require("http");
3. const server = http.createServer((req, res) => {
4. const url = req.url;
5. if (url === "/") {
6. res.write("<html>");
7. res.write("<head><title>Enter Message</title></head>");
8. res.write(
9. '<body><form action="/message" method="POST" name="message"><input type="text" /> <hr/> <button type="submit" >Send</button></form></body>'
10. );
11. res.write("</html>");
12. return res.end();
13. }
14. // process.exit()
15. res.setHeader("Content-Type", "text/html");
16. res.write("<html>");
17. res.write("</html>");
18. res.end();
19. });
21. server.listen(3000);

**3. Understanding the Basics**

33. Redirecting Requests

მომხმარებლის გადამისამართებაც შეგვიძლია ასევე, რომელშიც დაგვჭირდება სტატუსკოდი და ჰედერი, რომელიც ლოკაციას განსაზღვრავს. ჩვენს მაგალითში ფაილსაც ვიანხავთ კონკრეტულ ლოკაციაზე.

**app.js:**

1. if(url === '/message' && method === 'POST'){
2. fs.writeFileSync('user-message.txt', 'DUMMY')
3. res.statusCode = 302;
4. res.setHeader('Location', '/');
5. return res.end()
6. }

**3. Understanding the Basics**

34. Parsing Request Bodies

იმისათვის, რომ შემომავალი დეითა დავიჭიროთ, უნდა გამოვიყენოთ **req**-ზე არსებული ***.on()***მეთოდი. იგი პირველ არგუმენტად იღებს მოქმედების სტრინგს, ხოლო მეორედ ქოლბაქ ფუნქციას, ხოლო ფუნქცია რას დააბრუნებს, მოქმედების ტიპზეა დამოკიდებული.

ჩვენ შემთხვევაში ვიყენებთ **.on('data')** და **.on('end')** ფუნქციებს.

**.on('data')**ჩანკებად დაიჭერს შემოსულ ინფორმაცია, ამიტომაც იგი ნელ-ნელა შემოდის. მის ქოლბაქ ფუნქციაში დაბრუნდება თითო ჩანკი, რომლებიც სასურველია გარეთ არსებულ ცვლადში შევინახოთ, კარგი იქნება მასივში.

**.on('end')** ფუნქცია კი გამოიძახება, როცა **.on('data')** დასრულდება და მანდ შეგვეძლო ნოუდში ჩაშენებული მეთოდის -**Buffer.concat(ცვლადი).toString()** (ამ შემთხვევაში) გამოვიყენოთ, რომელიც გააერთიანებს ცვლადის მასივში არსებულ ბაფერებს და სტრინგის სახით გამოიტანს.

*კოდი შემდეგში...*

**3. Understanding the Basics**

34. Parsing Request Bodies

**app.js**-ის მონაკვეთი:

1. const body = [];
3. req.on('data', (chunk) => {
4. console.log(chunk)
5. body.push(chunk);
6. })
8. req.on('end', ()=>{
9. const parsedBody = Buffer.concat(body).toString();
10. const message = parsedBody.split('=')[1];
11. fs.writeFileSync('user-message.txt', message)
12. });

**3. Understanding the Basics**

36. Blocking and Non-Blocking Code

განსხვავება **fs.writeFileSync()**-სა და **fs.writeFile()**-ს შორის არის ის, რომ უკანასკნელი ასინქრონულად გაეშვება, ანუ მის შემდგომ მდგარი კოდი ეგრევე შესრულდება, მიუხედავად იმისა, ფუნქციას მუშაობა დასრულებული აქვს თუ არა. ეს უკანასკნელი მესამე არგუმეტნსაც იღებს, რომელიც ასევე ერორ ჰენდლინგისთვისაც გამოიყენება. სასურველია მაგ ქოლბაქის (მესამე არგუმენტის) ბლოკში უნდა ჩავწეროთ ის ლოგიკა, რომელიც ფაილის შექმნის მერე გვსურს რომ მოხდეს.

app.js მონაკვეთი:

1. const body = [];
3. req.on("data", (chunk) => {
4. console.log(chunk);
5. body.push(chunk);
6. });
8. req.on("end", () => {
9. const parsedBody = Buffer.concat(body).toString();
10. const message = parsedBody.split("=")[1];
11. fs.writeFile("user-message.txt", message, (err) => {
12. res.statusCode = 302;
13. res.setHeader("Location", "/");
14. return res.end();
15. });
16. });

**3. Understanding the Basics**

38. Using the Node Modules System

ნოუდში დაექსპორტების რამდენიმე მეთოდია, მათ შორისაა შემდეგნი:

1. module.exports = requestHandler;
3. module.exports = {
4. hendler: requestHandler,
5. someText: 'Some Hard Coded text'
6. };
8. module.exports.handler = requestHandler;
9. module.exports.someText = 'Some Hard Coded text';

**4. Improved Development Workflow and Debugging**

43. Installing 3rd Party Packages

**package.json**-ში **scripts**-ის სექციაში გაწერილი ფროფერთი-ვალიუ პეერებიდან სტანდარტულად ეშვება მაგალითად start ბრძანება, ანუ

"start": "node app.js"

**npm start**

თუმცა:

"start-app": "node app.js"

**npm run start-app**

**4. Improved Development Workflow and Debugging**

43. Installing 3rd Party Packages

**npm**-ის პაკეჯის ინსტალაციისას საჭიროა ვიცოდეთ:

**--save** - ლოკალურად, თუმცა პროდაქშენშიც გადავა

**--save-dev** - ლოკალურად, თუმცა არ გადავა პროდაქშენში

**-g**- გლობალურად.

მაგ: **npm install nodemon --save-dev**

**4. Improved Development Workflow and Debugging**

54. Changing Variables in the Debug Console

ამ სექციაში **debagger**-ზე ვიყავით ორიენტირებული, რომელიც **VsCode**-შია ჩაშენებული და დებაგინგში გვეხმარება.

მას კონკრეტულად არ ვაკონსპექტებ, თუმცა თუ საჭირო გახდა, თავიდან ვუყურებ და ჩავიწერ ყველაფერს...

დებაგირების წყაროები:

**NodeJS**-*https://nodejs.org/en/docs/guides/debugging-getting-started/*

**VScode**- *https://code.visualstudio.com/docs/nodejs/nodejs-debugging*

**5. Working with Express.js**

59. Installing Express.js

**express.js**გვთავაზობს მიდლვეარ ლოგიკას **NodeJS**-ის ვებისთვის.

იგი ინსტალირდება: npm install --save express

და ძირითად ფაილში იმპორტირდება და ინახება ცვლადში, მაგალითად სახელად **express**.

მის გასააქტიურებლად, ანუ გამოსაძახებლად, ვიყენებთ ცვლადის შესაბამისად ბრძანებას **express()**, რომელსაც სხვა ცვლადში ვინახავთ, რომელიც საბოლოო ჯამში მთელ **express**წარმოადგენს თითქმის.

**app.js:**

1. const http = require("http");
3. const express = require('express');
5. const app = express();
7. const server = http.createServer(app);
9. server.listen(3000, () => {
10. console.log('server is runing')
11. });

**5. Working with Express.js**

60. Adding Middleware

**app**-ის ერთ-ერთი მიდლვეარია **.use(),** რომელში ჩასმული ფუნქციაც ყოველ ჯერზე გაეშვება, როცა რექვესტი შემოვა.

მას თუ ფუნქციას ჩავაწვდით, აღნიშნული ფუნქცია მიიღებს სამ არგუმენტს: **req, res, next**, საიდანაც:

**req - შემოსულ რექვესტებს ჰენდლავს;**

**res - გასაგზავნ პასუხს;**

**next - რომელიც საბოლოო ჯამში ფუნქციის სახით გამოიძახება - next() - საშუალებას აძლევს რექვესტს ერთი მიდლვეარიდან მეორეში გადავიდეს.**

**app.js მონაკვეთი:**

1. app.use((req, res, next)=>{
2. console.log('In the middleware!')
3. next()
4. })
6. app.use((req, res, next)=>{
7. console.log('In another middleware!')
8. // ...
9. })

**5. Working with Express.js**

61. How Middleware Works

**res**-ს აქვს თავის იფუნქციები, მათ შორის ძველებიც, თუმცა ექსპრესი ამატებს ერთ უნიკალურ ფუნქციას - **.send()**, რომელიც **any**ტიპისაა და შეგიძლია ნებისმიერი ტიპის რექვესტი გაგზავნო.

ასევე ექსპრესი აბამს ჰედერებს, თუმცა გადაწერა შეიძლება.

მაგ.:

1. app.use((req, res, next)=>{
2. res.send('<h1>Hello from Express</h1>')
3. })

**5. Working with Express.js**

62. Express.js - Looking Behind the Scenes

**http.createServer(app)-ის ალტერნატივად განიხილება შემდეგი:**app.listen(3000)

მაგალითად, თუ ადრე ვიყენებდით ამას:

1. const server = http.createServer(app);
2. server.listen(3000, () => {
3. console.log('server is runing')
4. });

ახლა გამოვიყენებთ:

1. app.listen(3000)

**5. Working with Express.js**

63. Handling Different Routes

**app.use()** იღებს რამდენიმე არგუმენტს გარდა ქოლბაქისა, რომელიც სავალდებულოა.

მისი პირველი არგუმენტი შეიძლება იყოს **path**-ი, რომელზე შემოსულ რექვესტსაც უპასუხებს.

მაგალითად, თუ გამოყენებულია**"/"** - მსგავსი **path**-ი, იგი მაინც ყვლეა ტიპის **path**-ს უპასუხებს, რადგან აღნიშნული ადგენეს, შეიცავს თუ არა ლინკი კონკრეტულ სინტაქსს, **"/"** - კი ყველა ლინკშია, ამიტომაც მნიშვნელოვანია თანმიმდევრობა აღნიშნული მიდლვეარების, რათა სწორად გადამისამართდეს რექვესტი, რამდენადაც რექვესტის ჰენდლინგისას ყოველთვის **.next()** ფუნქციას არ გამოვიყენებთ.

**app.js:**

1. const express = require('express');
2. const app = express();
4. app.use('/product', (req, res, next)=>{
5. console.log('In another middleware!')
6. res.send('<h1>The "ADD PRODUCT" page</h1>')
7. })
9. app.use('/', (req, res, next)=>{
10. console.log('In another middleware!')
11. res.send('<h1>Hello from Express</h1>')
12. })
14. app.listen(3000)

**5. Working with Express.js**

64. Parsing Incoming Requests

წინაზე იმისათვის, რომ გადამისამართება მოგვეხდინა, სტატუს კოდი და ლოქეიშენ ჰედერები დაგვჭირდა, ახლა კი აღნიშნული ძალიან მარტივად ხდება, უბრალოდ პასუხად ვაბრუნებთ **.redirect()** მეთოდს, რომელსაც არგუმენტად ეთითება გადამისამართების მიმართულება.

1. app.use('/product', (req, res, next)=>{
2. res.redirect('/')
3. })

ასევე, ვიცით რომ არგუმენტი **req**- ლოგკურად კითხულობს შემოსულ რექვესტს. თუ ვეცდებით, რომ **req.body**-ის მივწვდეთ, ანდიფაინდი იქნება საწყისდზე, რადგან არ ვახდენთ შემოსული დეითას პარსირებას.

ამისთვის გამოიყენება ცალკე პაკეჯი, რომელსაც ვაინსტალირებთ შემდეგი ბრძანებით:

npm install --save body-parser

შემდეგ კი, როუტის მიდლვეარებამდე ვადეკლარირებთ (ანუ ყველა როუტს უნდა უსწრებდეს):

1. const bodyParser = require('body-parser')
2. app.use(bodyParser.urlencoded())

თუმცა, რამდენადაც express-ის მეოთხე ვერსიიდან შიგნითვე ჩაიშენა ეს ფუნქცია, გამოვიყენებ შემდეგნაირად (body-parser აღარაა საჭირო):

1. app.use(express.urlencoded({extended: true}))

**5. Working with Express.js**

64. Parsing Incoming Requests

**urlencoded({extended: true})** - შემოსულ დეითას, რომელიც ძირითადად შემოდის ფორმსიდან, ოღონდ აღნიშნული **url**-ში უნდა მიდიოდეს, ანუ ინფუთებს ნეიმები უნდა ჰქონდეთ და ასევე ფორმს მეთოდი პოსტი. საერთო ჯამში, აღნიშნული **url**-ს კითხულობს.

**5. Working with Express.js**

65. Limiting Middleware Execution to POST Requests

მიდლვეარს ასევე შეგვიძლაი განვუსაზღვროთ - რომელი ტიპის რექვესტებს მოუსმინოს, ამისათვის **.use()** უბრალოდ უნდა ჩავანაცვლოთ მაგალითად - **.get()**ან**.post()**და ა.შ.

1. app.get('/app-product', (req, res, next)=>{
2. res.send('<form action="/product" method="POST"> <input type="text" name="title" /> <button type="submit">Add Product</button> </form>')
3. })
5. app.post('/product', (req, res, next)=>{
6. console.log(req.body)
7. res.redirect('/')
8. })
10. app.get('/', (req, res, next)=>{
11. res.send('<h1>Hello from Express</h1>')
12. })

**5. Working with Express.js**

66. Using Express Router

როუტების ორგანიზებისთვის საუკეთესო გზაა ისინი დავსპლიტოთ სხვადასხვა ფაილებში და იმ კონკრეტულ ფაილებში დავაიმპორტოთ ექსპრესი და ვისარგებლოთ მასში არსებული ფუნქციით - **.Router()**, რომელსაც შევინახავთ ცვლადში და **app**-ს **router**-ით ჩავანაცვლებთ, ხოლო ძირითად ფაილში უბრალოდ **app.use()**-ში ჩასმული დაიმპორტებული როუტიც საკმარისი იქნებოდა.

როუტის ფაილი (მაგ. **shop.js**):

1. const express = require('express');
2. const router = express.Router()
4. router.get('/', (req, res, next)=>{
5. res.send('<h1>Hello from Express</h1>')
6. })
8. module.exports = router;

ძირითად იფაილი **(app.js)**:

1. const express = require('express');
3. const app = express();
5. const shopRoutes = require('./routes/shop')
7. app.use(express.urlencoded({extended: true}))
9. app.use(shopRoutes);
11. app.listen(3000)

**5. Working with Express.js**

67. Adding a 404 Error Page

ნოთ ფოუნდის შემთხვევაში, შეგვიძლია პირდაპირ **app.use()** გამოვიყენოთ ძირითად ფაილში, რომელიც ნებისმიერი ტიპის რექვესტს დაჰენლდავს, რომელიც ზემოთ მოცემულ მიდლვეარებში ვერ დავიჭირეეთ და ასევე, როგორც სხვა რესპონსებში, აქაც შეგვიძლია გამოვიყენოთ**.send()**-ამდე მაგრამ მას მიბმული მეთოდი - **.status()** - რომელიც არგუმენტად სტატუს-კოდს იღებს.

1. app.use((req, res, next)=>{
2. res.status(404).send('<h1>404 - Page not found</h1>')
3. })

**5. Working with Express.js**

68. Filtering Paths

იმ შემთხვევაში, თუ კონკრეტულ როუტში მოქცეულ **path**-ებს საერთო დასაწყისი აქვთ, შეგვიძლია ძირითად ფაილში ეგ საერთო საწყისი გავწეროთ **app.use()**-ის პირველ არგუმენტად, ხოლო თავად როუტში მხოლოდ სხვაობები ჩავწეროთ.

**shop.js:**

1. router.get('/app-product', (req, res, next)=>{
2. res.send('<form action="/admin/add-product" method="POST"> <input type="text" name="title" /> <button type="submit">Add Product</button> </form>')
3. })

**app.js:**

1. app.use('/admin', adminRoutes);

**5. Working with Express.js**

70. Serving HTML Pages

**HTML**ფაილების გასაგზავნად, სასურველია ჯერ ცალკე ფოლდერი შევქმნათ, მაგალითად views ფოლდერი, რომელშიც შევინახავთ **html**ფაილებს.

შემდეგ კი რომელ **path**-შიც გვსურს **.send()**-ის ნაცვლად შეგვიძლაი გამოვიყენოთ **.sendFile()**, რომელიც მოითხოვს აბსოლუტურ **path**-ს, თუმცა იგი იწყება რუტ ფოლდერს უმიზნებს ავტომატურად, რომელიც ჩვენს ოპერაციულ სისტემაში იმყოფება, ამიტომაც უბრალოდ ვერ დავწერთ - res.sendFile('/views/shop.html')-ს.

ამის ნაცვლად, უნდა შემოვიტანოთ ძრავის მოდული - **path**- და გამოვიყენოთ მისი ფუნქცია**.join()**, რომელიც პირველ არგუმენტად მიიღებს ნოუდში ჩაშენებულ გლობალური წვდომის ცვლადს -**\_\_dirname** - რომელიც მიუთითებს ჩვენი პროექტის დირექტორიაზე, მეორე არგუმენტად, რამდენადაც ჩვენ შვილ ფოლდერიდან ვიძახებთ ამას, ერთი ფოლდერით უკან გადავიდეთ და ამისთვის ჩავაწვდით - '**../**', მესამეში კი უკვე ფოლდერის სახელი და მეოთხეში - ფაილის, რომელსაც ვიძახებთ.

**5. Working with Express.js**

70. Serving HTML Pages

**\*შენიშვნა:** ჩვენ შეგვეძლო, რასაც **.join()** აკეთებს ხელითაც გაგვეკეთებინა, ანუ **/**-ებით და ქონქათენეიშენით, თუმცა არაა სასურველი, რადგან **\_\_dirname** და**.join()** ასევე ცნობს სხვადასხვა ოპერაციულ სისტემების ფაილურ სისტემას და ისე ადგენს კონკრეტული პრეოქტის დირექტორიას, მაგალითად, ზოგ შემთხვევაში **path**იწერება **/** -ასე, ზოგში -**\** - ასე.

**shop.js:**

1. router.get('/', (req, res, next)=>{
2. res.sendFile(path.join(\_\_dirname, '../' ,'views', 'shop.html'))
3. });

**5. Working with Express.js**

73. Using a Helper Function for Navigation

**\*შენიშვნა:**სასურველია**'../'**- ის ნაცვლად გამოვიყენო მხოლოდ **'..'**მაგალითად...

ასევე, დირექტორია შეგვიძლია ცალკეც დავადეკლარიროთ და შემდეგ ყველგან ეგ მოვიხმაროთ.

ამისთვის, ცალკე ფოლდერს და მასში ფაისლ შევქმნი, მაგალითად, **path.js**-ის სახელით, შიგნით კი შემდეგი კოდი:

1. const path = require('path');
2. module.exports = path.dirname(process.mainModule.filename)

ხოლო სხვაგან კი უბრალოდ დავაიმპორტებთ. აღნიშნული **'..'**-საც ჩაანაცვლებს, მაგალითად:

ადრე იყო:

1. router.get('/app-product', (req, res, next)=>{
2. res.sendFile(path.join(\_\_dirname, '..', 'views', 'add-product.html'))
3. })

ახლა არის:

1. const rootDir = require('../util/path');
2. router.get('/app-product', (req, res, next)=>{
3. res.sendFile(path.join(rootDir, 'views', 'add-product.html'))
4. })

**5. Working with Express.js**

75. Serving Files Statically

თუ გვსურს, რომ **css**და **js**ფაილებსაც მოვემსახუროთ, რომლებიც მომხმარებლის მხარეს ჩამოიტვირთებიან, სასურველია ძირითად ფაილში ექსპრესს მივუთითოთ, თუ სტატიკურად რომელ ფოლდერს მოემსახუროს.

ამისთვის თავად **express**-ში არსებობს მეთოდი **.static()**, რომელიც მიიღებს არგუმენტად იმ ფოლდერის მისამართს, რომელსაც უნდა მოემსახუროს (მისამართის მიმაგრების სინტაქსი იგივეა).

იგი ასევე,  მაგალითად, თუ **html**-ი, რომელიც **.sendFile()**-ით გავგზავნეთ, გაილინკავს **css**-ს, რომელიც სულ სხვა ფოლდერში იმყოფება, მაგალითად **public/css**-ში, ხოლო თავად ეს **html views**-ის ფოლდერში, მაინც გადაამისამართებს იმ დირექტორიში, რომელსაც სტატიკურად ემსახურება.

*კოდის მაგალითი:*

**app.js:**

1. app.use(express.static(path.join(\_\_dirname, 'public')));

**shop.html:**

1. <link rel="stylesheet" href="/css/main.css">

**5. Working with Express.js**

75. Serving Files Statically

ასევე შეგვეძლო მრავალი სტატიკური ფოლდერი დაგვედეკლარირებინა, შედეგად კი ყველგან შეამოწმებდა აღნიშნულ ფაილს, მაგ:

1. app.use(express.static(path.join(\_\_dirname, 'public')));
2. app.use(express.static(path.join(\_\_dirname, 'me')));
3. app.use(express.static(path.join(\_\_dirname, 'other')));
4. app.use(express.static(path.join(\_\_dirname, 'guys')));

**6. Working with Dynamic Content & Adding Templating Engines**

79. Sharing Data Across Requests & Users

რაიმეს ექსპორტირებისთვის კიდევ არსებობს შემდეგი გზა:

**exports.routes = router**

სადაც ობიექტი გადის გარეთ და შემდეგ მისი მიღება ხდება შემდეგნაირად:

1. const adminData = require('./routes/admin');
3. app.use('/admin', adminData.routes);

ამ სექციაში უბრალოდ დეითას დროებით ვინახავთ სერვერსაიდ ჯავასკრიპტის მასივში, რომელიც ამ დეითას სხვა დაქონექთებულ მომხმარებლებს უზიარებს.

**6. Working with Dynamic Content & Adding Templating Engines**

81. Installing & Implementing Pug

**Template engines** - გვეხმარებიან დინამიურად შევყაროთ დეითა **HTML**-ში.

*ისინი განხილული არიან ფოტოზე...*

სამივეს დასაინსტალირებლად ვუშვებთ შემდეგ ბრძანებას:

npm install --save ejs pug express-handlebars

***express****- დამატებაა, რადგან მის ბიბლიოთეკას ვიყენებთ და მარტივად ადაპტირებადია კონკრეტული თემფლეით ენჯინისთვის.*

ახლა გამოვიყენებთ **pug**-ს, თუმცა სანამ დავიწყებდეთ მის გამოყენებას, ჯერ გავარკვიოთ რას აკეთებს **app.set()** მეთოდი, რომელიც ექსპრესის ნაწილია:

იგი საშუალებას გვაძლევს ნებისმიერი ტიპის მნიშვნელობის დაყენებას ჩვენს აპლიკაციაში გლობალურად, რომლებიც ექსპრესს რომც არ ესმოდეს რა არის, ჩვენ მაინც შევძლებთ მის წაკითხვას **app.get()**-ის საშუალებით.

მასში პირველ არგუმენტად ვაწვდით ფროფერთის, ხშირად სტრინგი, რომელიც განსაზღვრავს კონკრეტულ ქცევას, მაგალითად, იმ ქცევას, რომელიც მეორე არგუმენტზე იქნება დამოკიდებული.

**6. Working with Dynamic Content & Adding Templating Engines**

81. Installing & Implementing Pug

ერთ-ერთი მათგანია**'views'** ან**'view engine'**;

**app.set('view engine', 'pug')** - უთითებს ძრავს, რომელიც უნდა გამოიყენოს;

**app.set('views', 'views')** - უთითებს გამსოაყენებელი თემფლეითების ლოკაციას (დეფაულტზე თუ რუთ ფოლდერშია ეგ ფოლდერი, ეს არაა საჭირო);

შემდეგ კი ვადეკლარირებთ კონკრეტულ ფოლდერში (რომელიც ზევით მივუთითეთ), **html**-ის ნაცვლად **pug**გაფართოების ფაილს: **shop.pug**, სადაც განსხვავებული სინტაქსით იწერება html-ი, თუმცა საბოლოო ჯამში პაგის ძრავი მას ნორმალურ html-ში გადაიყვანს.

*გაგრძელდება...*

**6. Working with Dynamic Content & Adding Templating Engines**

81. Installing & Implementing Pug

**მაგ:**

1. link(rel="stylesheet", href="/css/product.css")
2. body
3. header.main-header
4. nav.main-header\_\_nav
5. ul.main-header\_\_item-list
6. li.main-header\_\_item
7. a.active(href="/") Shop

ასევე, მნიშვნელოვანია, რომ **.sendFile()**-ის ნაცვლად, გამოვიყენოთ**.render()** ფუნქცია, რომელიც ასევე ექსპრესს ეკუთვნის და არგუმენტად იღებს ფაილის სახელს, გაფართოება აღარაა საჭირო, რადგან იგი უკვე დეკლარირებულია დეფაულტ თემფლეით ეჯნინის გაფართოებაში ზევით.

**6. Working with Dynamic Content & Adding Templating Engines**

82. Outputting Dynamic Content

მას შემდეგ, რაც **.render()** ფუნქცია გაიმართება, pug ფაილში შეგვიძლია კონკრეტულ ობიექტს მივწვდეთ მარტივი სინტაქსით: **#{docTitle}**, სრულად:

1. title #{docTitle}

ანუ **js**ფაილი შემდეგნაირადაა:

1. router.get('/', (req, res, next)=>{
2. const products = adminData.products;
3. res.render('shop', {
4. prods: products,
5. docTitle: 'Shop'
6. });
7. });

ხოლო **pug**ფაილში კი:

1. title #{docTitle}

**6. Working with Dynamic Content & Adding Templating Engines**

82. Outputting Dynamic Content

ხოლო იმ შემთხვევაში, თუ ლუპია გასაკეთებელი, pug-ში შემდეგნაირად ვწერთ:**each product in prods** (ჩვენი ობიექტის ფროფერთის შესაბამისად);

**shop.pug:**

1. .grid
2. each product in prods
3. article.card.product-item
4. header.card\_\_header
5. h1.product\_\_title #{product.title}

**shop.js:**

1. router.get('/', (req, res, next)=>{
2. const products = adminData.products;
3. res.render('shop', {
4. prods: products,
5. docTitle: 'Shop'
6. });
7. });

**6. Working with Dynamic Content & Adding Templating Engines**

82. Outputting Dynamic Content

**PUG**-ში გასათვალისწინებელია, რომ შვილობილები იყვნენ მშობელთან შედარებით, უფრო შეწეულები, ანუ შემდეგნაირად:

1. main
2. .grid
3. each product in prods
4. article.card.product-item
5. header.card\_\_header
6. h1.product\_\_title #{product.title}
7. .card\_\_image
8. img(src="https://cdn.pixabay.com/photo/2016/03/31/20/51/book-1296045\_960\_720.png" alt="A Book")
9. .card\_\_content
10. h2.product\_\_price $19.99
11. p.product\_\_description A very interesting book about so many even more interesting things!
12. .card\_\_actions
13. button.btn Add to Cart

**6. Working with Dynamic Content & Adding Templating Engines**

82. Outputting Dynamic Content

ხოლო თუ ქონდიშენალ ლოგიკის დაწერა გვსურს **PUG**-ში:

1. main
2. if prods.length > 0
3. h1 they are
4. else
5. h1 they are not

**6. Working with Dynamic Content & Adding Templating Engines**

85. Adding a Layout

ჩვენ ასევე შეგვიძლია გვქონდეს ერთი კონკრეტული ლეიაუტი, რომელსაც სხვა დინამიურ pug ფაილებში გამოვიყენებთ, მაგალითად, დინამიური ლეიაუთი, რომელიც ერთხელ შეიქმნება მხოლოდ, იქნება დაახლოებით ასეთი:

1. doctype html
2. html(lang="en")
3. head
4. ...
5. block styles
6. body
7. ...
8. block content

სადაც ქივორდი block გვაძლევს საშუალებს, რომ განვსაზღვროთ კონკრეტულ ლოკაციაზე ის, რომ აქ შეიძლება სხვა **pug**ფაილიდან, სადაც ეს ექსთენდდება, სხვა თემფლეითიც ჩაემატოს.

*გრძელდება...*

**6. Working with Dynamic Content & Adding Templating Engines**

85. Adding a Layout

მისი გამოყენება შემდეგნაირად ხდება:

1. extends layouts/main-layout.pug
3. block styles
4. link(rel="stylesheet", href="/css/forms.css")
5. link(rel="stylesheet", href="/css/product.css")
7. block content
8. main
9. form(action="/admin/add-product", method="post").product-form
10. .form-control
11. label(for="title") Title
12. input(type="text" name="title")#title
13. button(type="submit").btn Add Product

სადაც **extends**ქივორდის მერე ვუთითებთ ლეიაუთის ლოკაციას და ფაილს, ხოლო სადაც **block**ქივორდი გვქონდა, მისი სახელითურთ, ვუწერთ კონკრეტულ კონტენტს.

**6. Working with Dynamic Content & Adding Templating Engines**

86. Finishing the Pug Template

ქონდიშენალ სტეიტმენტი ტერნერი ოპერატორთან ერთად შეგვიძლია ასევე გამოვიყენოთ **class**ქივორდში, როცა მას ფრჩხილებს გავუტოლებთ, მაგალითად:

**main-layout.pug:**

1. a(href="/admin/app-product", class=(path==='/admin/app-product' ? 'active' : '')).main-header\_\_item Add Product

**path**კი მოდის - **admin.js**:

1. router.get('/app-product', (req, res, next)=>{
2. res.render('add-product', {pageTitle: 'Add Product', path: '/admin/app-product'})
3. });

ასევე შეგვიძლია დინამიურ ლეიაუტში **block**ქივორდის გარდა გავმოვიყენოთ სტანდარტული**#{}** ცვლადის ჩასასმელი სინტაქსი, ჩავაწოდოთ რაიმე და რომელი თემფლეითიც დაარენდერებს ამ ლეიაუტს, მისი ობიექტიდან უნდა ჩააწოდოს იგივე სახელის ცვლადი ანუ ფროფერთი.

1. title #{pageTitle}

**6. Working with Dynamic Content & Adding Templating Engines**

86. Finishing the Pug Template

**PUG**-ის დოკუმენტაცია:

*https://pugjs.org/api/getting-started.html*

**6. Working with Dynamic Content & Adding Templating Engines**

88. Working with Handlebars

**Handlebars**- არის თემფლეითი ენჯინის კიდევ ერთი პაკეტი.

ცვლილების გამო, აღნიშნული ბრძანება თავიდან და შემდეგნაირად უნდა გავუშვა **CLI**-დან:

npm install --save express-handlebars@3.0

**PUG**-ისგან განსხვავებით **express-handlebars** არ არის ჩაშენებული ენჯინი, ამიტომაც საწყისში მისი დაიმპორტება გვიწევს და ასევე **.set()**-ის ნაცვლად უნდა გამოვიტენოთ - **.engine()** მეთოდი, რომელიც პირველ არგუმენტად იღებს ენჯინი სახელს (ჩვენზეა), უბრალოდ არ უნდა გადაკვეთოს სხვა ენჯინის სახელი, ხოლო მეორე არგუმენტად იმ ცვლადს, რომელშიც დაიმპორტებული ჰენდლბარი შევინახეთ, როგორც ფუნქციას.

1. const expressHbs = require('express-handlebars');
2. app.engine('handlebars', expressHbs())

აი, ახლა კი გამოვიყენებთ**.set()**მეთოდს და **'view engine'**-საც გადავცვლით, საბოლოო ჯამში კი:

1. const expressHbs = require('express-handlebars');
3. app.engine('handlebars', expressHbs())
5. app.set('view engine', 'handlebars');
6. app.set('views', 'views');

**6. Working with Dynamic Content & Adding Templating Engines**

88. Working with Handlebars

ასევე, მნიშვნელოვანი: რაც **.engine()**-ში დავარქვით, იგივე გაფართოება უნდა ჰქონდეთ ფაილებსაც, ამიტომ მათ თუ გადავაკეთებთ **hbs**-ზე, უკეთესი იქნება:

1. app.engine('hbs', expressHbs())
2. app.set('view engine', 'hbs');

**6. Working with Dynamic Content & Adding Templating Engines**

88. Working with Handlebars

**handlebars**-ის შემთხვევაში, სტანდარტული **html**-ი გამოიყენება, ხოლო ობიექტის ფროფერთიები ანგულარის მსგავსად, ჩაეწოდება ორმაგი ფიგურულ ფრჩხილში ჩაწოდებული, მაგალითად:

1. <title>{{ pageTitle }}</title>

**6. Working with Dynamic Content & Adding Templating Engines**

89. Converting our Project to Handlebars

**handlebars**აქვს განსაკუთრებული მიდგომა ასევე ქონდიშენალ და ლუპის ლოგიკასთან, რომელიც თემფლეითში ეშვება.

ქონდიშენალის შემთხვევაში, გვჭირდება ჰაშთაგი და **if**, შემდეგ ცვლადი (იგივე ფროფერთი), რომელიც ბულიანს აბრუნებს და იწერება ობიექტში და მასში მოქცეული **html**კოდი და შემდეგ დამხურავი **/if**, ასევე შიგნით შეგვიძლია უბრალოდ **else**მოვაქციოთ, რომელსაც ქვევით ასევე ექნება **html**კოდი.

მაგალითი:

1. {{#if hasProducts }}
2. <div>...</div>
3. {{else}}
4. <h1>no products found</h1>
5. {{/if}}

ლუპის ლოგიკისას კი იგივენაირად ოღონდ **each**ქივორდის გამოყენებით, რომელიც მხოლოდ ცვლადს (იგივე ფროფერთის) დაუმიზნებს, სადაც მასივი ინახება, ხოლო თავად შიგნით **html**-ში this. - ქივორდით მივწვდებით ობიექტში არსებულ ფროფერთიებს. ისიც მარტივად **/each**-ით დაიხურება.

მაგალითი:

1. {{#each prods}}
2. <h1>{{ this.title }}</h1>
3. {{/each}}

**6. Working with Dynamic Content & Adding Templating Engines**

90. Adding the Layout to Handlebars

**handlebars**-ში ლეიაუტის დეკლარირება ცოტა სხვანაირად ხდება. იგი ჯერ **.engine()**მეთოდში მეორე არგუმენტად გამოძახებულ ფუნქციას ობიექტად უნდა ჩაეწოდოს და დაკონფიგურირდეს **layoutsDir**, რომელიც მისამართი იქნება ლეიაუთების (დეფაულტზე **layouts**საკმარისია), ასევე **defaultLayout**, რომელშიც იქნება სახელი იმ ლეიაუთის, რომელიც ყველგან გამოიყენება, ასევე მნიშნველოვანია ჰქონდეს კიდევ ერთი ფროფერთი - **extname**, რომელიც განსაზღვრავს ლეიაუტის გაფართოებას, რომელიც დინამიურად უნდა ჩაეწოდოს, ჩვენს შემთხვევაში ეს **'hbs'** იქნება.

მაგ. - **app.js:**

1. app.engine('hbs', expressHbs({layoutsDir: 'views/layouts', defaultLayout: 'main-layout', extname: 'hbs'}))

**6. Working with Dynamic Content & Adding Templating Engines**

90. Adding the Layout to Handlebars

ხოლო რაც შეეხება დინამიური ლეიაუტის შექმნას, იგი სტანდარტულად შეიქმნება, ანუ ცალკე **.hbs** გაფართოების ფაილი, რომელშიც გარკვეული **html**კოდი წავა.

მასში დასარენდერებელი დინამიური დეითა, მაგალითად **link**-ების დარენდერება ფროფერთიებით გაკონტროლდება.

რაც შეეხება დინამიურ კონტენტს, აღნიშნული გაკონტროლდება სამმაგი ფიგურული ფრჩხილით - **{{{ body }}} - ყველა სხვა .hbs ფაილი ამის მერე body-ში დარენდერდება.**

ხოლო რაც შეეხება რაიმე ფროფერთის, მაგალითად კლასების შინაარსის პირობითობას, ისიც ასევე ქონდიშენალის სტანდარტული სინტაქსით გაკონტროლდება.

*მაგალითები შემდეგში...*

**6. Working with Dynamic Content & Adding Templating Engines**

90. Adding the Layout to Handlebars

**link-ების დინამიური კონტროლი**

**layout.hbs:**

1. <link rel="stylesheet" href="/css/main.css" />
2. {{#if formsCSS}}
3. <link rel="stylesheet" href="/css/forms.css" />
4. {{/if}}
5. {{#if productCSS}}
6. <link rel="stylesheet" href="/css/product.css" />
7. {{/if}}

**shop.js:**

1. router.get('/', (req, res, next)=>{
2. ...
3. res.render('shop', {
4. ...
5. productCSS: true,
6. });
7. });

**დინამიური კონტენტის რენდერი:**

**layout.hbs:**

1. <body>
2. {{{body}}}
3. </body>

**ფროფერთიების ქონდიშენალინგი:**

**layout.hbs:**

1. <li class="main-header\_\_item"><a class="{{#if activeShop }}active{{/if}}" href="/">Shop</a></li>

**shop.js:**

1. res.render('shop', {
2. ...
3. activeShop: true,
4. ...
5. });

**6. Working with Dynamic Content & Adding Templating Engines**

90. Adding the Layout to Handlebars

**Handlebars**-ის ბიბლიოთეკა:

*https://handlebarsjs.com/*

**6. Working with Dynamic Content & Adding Templating Engines**

91. Working with EJS

**EJS**-სს არ სჭირდება**.engine()**მეთოდის გამოყენება, იგი **PUG**-ის მსგავსად, მხოლოდ**.set()** ფუნქციაში მეორე არგუმენტად ჩაეწოდება და ეგრე დეკლარირდება.

იგი **PUG**-ის მსგავსად, ქონდიშენალ ლოგიკას თემფლეითშივე იყენებს და **HBS**-ის მსგავასდ, ნორმალური **HTML**-ის მსგავსი არის.

ხოლო, რაც შეეხება ჯავასკრიპტის თემფლეითში წერის სინტაქსს, მარტივია, უბრალოდ მცირედით განსხვავებული.

ცვლადი შემდეგნაირად შემოგვაქვს:

1. <title><%= pageTitle %></title>

ქონდიშენალს შემდეგნაირად ვწერთ:

1. <% if(prods.length > 0) { %>
2. <div>
3. ...
4. </div>
5. <% } else { %>
6. <h2>Not found</h2>
7. <$ } %>

ლუპს შემდეგნაირად ვაკეთებთ (თუმცა, რამდენადაც ვანილა js-ში შეგვიძლია ვწეროთ, მასივზე მოქმედების ლუპების შეგვიძლია გამოვიყენოთ):

1. <% for(let product of prods) { %>
2. <div>
3. <h1>product.title</h1>
4. </div>
5. <% } %>

**6. Working with Dynamic Content & Adding Templating Engines**

92. Working on the Layout with Partials

**PUG**-სა და **HBS**-ისგან განსხვავებით, **EJS**ლეიაუტებს არ იყენებს, რაც იმას ნიშნავდა, რომ კონკრეტულ ლეიაუტში დაგვერენდერებინა კონკრეტული როუტები, ამის ნაცვლად, ეს გამოიყენებს ე.წ. **INCLUDES**, რომელიც შექმნის ფაილებს, რომლებიც სხვადასხვა **HTML**-ში გამოიძახებიან და დარენდერდებიან.

**jead.ejs (includes file):**

უბრალოდ HTML კოდი

**მაგ. shop.ejs:**

1. <%- include('includes/head.ejs') %> //ანუ მისამართი იმ includes ფაილის

ასევე ფროფერთიებშიც იგივე სინტაქსით გავუწერდით ქონდიშენალ ლოგიკას:

1. <li class="main-header\_\_item"><a class="<%= path === '/' ? 'active' : '' %>" href="/">Shop</a></li>

**6. Working with Dynamic Content & Adding Templating Engines**

94. [OPTIONAL] Assignment Solution

***EJS docs:***

https://ejs.co/#docs

**7. The Model View Controller (MVC)**

97. What is the MVC?

**MVC - Models, Views, Controllers**

*დანარჩენი ფოტოშია...*

**7. The Model View Controller (MVC)**

98. Adding Controllers

**Views**- უკვე გამოვიყენეთ როგორც, თემფლეით ენჯინი, მაგალითად **EJS**;

**Controller**- წარმოადგენს თავად ფუნქციას, რომელიც ჩაშვებულია მიდლვეარში, მაგალითად:

1. app.use((req, res, next)=>{
2. ...
3. });

ამ კონკრეტული ფუნქციების სხვა ფაილში გადატანა, მოგვცემს საშუალებას უფრო ორგანიზებულად მივუდგეთ, ანუ **MVC**სწორად გამოვიყენოთ, შედეგად:

**products.js (controller):**

1. exports.postAddProduct = (req, res, next) => {
2. products.push({ title: req.body.title });
3. res.redirect("/");
4. };

**admin.js:**

1. const productsController = require('../controllers/products');
2. router.post("/add-product", productsController.postAddProduct);

**7. The Model View Controller (MVC)**

100. Adding a Product Model

**Models**- აღნიშნული კი საჭიროა ცალკე მოდელის შესაქმნელად, რომელიც სასურველია იყოს კლასი და მისი ინსტანსები შეგვიძლია გამოვიყენოთ სხვადასხვა დანიშნულებით.

მოდელიც ასევე ცალკე ფოლდერში იქმნება და შემდეგ სხვა ფაილების მიერ იმპორტდება.

**product.js (model):**

1. const products = [];
3. module.exports = class Product {
4. constructor(t) {
5. this.title = t;
6. }
7. save() {
8. products.push(this);
9. }
10. static fetchAll() {
11. return products;
12. }
13. };

**product.js:**

1. const Product = require('../models/product')
3. exports.postAddProduct = (req, res, next) => {
4. const product = new Product(req.body.title);
5. product.save();
6. ...
7. };
8. exports.getProducts = (req, res, next) => {
9. const products = Product.fetchAll()
10. ...
11. };

**7. The Model View Controller (MVC)**

102. Fetching Data from Files Via the Model

იმ შემთხვევაში, თუ გვენდომებოდა ფაილურ სისტემაში შეგვენახა ინფორმაცია, **save()**მეთოდს, რომელიც კლასში გვიწერია, შემდეგნაირად გადავაკეთებდით:

1. fs.readFile(p, (err, fileContent) => {
2. let products = [];
3. if (!err) {
4. products = JSON.parse(fileContent);
5. }
6. products.push(this);
7. fs.writeFile(p, JSON.stringify(products), (err) => {
8. console.log(err);
9. });
10. });

*გაგრძელდება...*

**7. The Model View Controller (MVC)**

102. Fetching Data from Files Via the Model

ხოლო**static fetchAll()**-ს კი:

1. static async fetchAll(cb) {
2. fs.readFile(p, async (err, fileContent) => {
3. try {
4. const file = await JSON.parse(fileContent);
5. cb(file)
6. } catch (err) {
7. cb([])
8. }
9. });
10. }

ხოლო რაც შეეხება გამოძახების, ანუ **fetchAll**-ის გამოყენებას, რამდენადაც ასინქრონულია, გამოძახების ფაილში შემდეგნაირად შევცვლიდით:

1. exports.getProducts = (req, res, next) => {
2. Product.fetchAll((products) => {
3. res.render("shop", {
4. prods: products,
5. pageTitle: "Shop",
6. path: "/",
7. hasProducts: products.length > 0,
8. activeShop: true,
9. productCSS: true,
10. });
11. });
12. };

**7. The Model View Controller (MVC)**

104. Wrap Up

**MVC ლინკი:**

*https://developer.mozilla.org/en-US/docs/Glossary/MVC*

**9. Dynamic Routes & Advanced Models**

120. Loading Product Detail Data

დინამიური ინფორმაციის მიღება **req**-იდან შეგვიძლია მასში ჩაშენებული ობიექტის, **params**-ის წყალობით, მაგალითად:

1. router.get('/products/:productId', (req, res, next) => {
2. const prodId = req.params.productId
3. console.log(prodId)
4. res.redirect('/')
5. });

**9. Dynamic Routes & Advanced Models**

122. Passing Data with POST Requests

იმ შემთხვევაში, თუ ისე აღმოჩნა, რომ **include**-ს ვაკეთებთ, ანუ სხვა თემფლეითი შემოგვაქვს **EJS**-ში, იქ, სადაც ლუპია და ლუპის ცვლადის სახელი ემთხვევა იმ ცვლადისას, რომელსაც ვიძახებთ, თავად **include**-ში უნდა ჩავამატოთ მეორე არგუმენტი ობიექტის სახით და მივუთითოთ, რომ მოცემული ცვლადი სხვა ფროფერთის უკავშირდება. მაგ:

1. <% for (let product of prods) { %>
2. <h1 class="product\_\_title"><%= product.title %></h1>
3. <div class="card\_\_actions">
4. <%- include('../includes/add-to-cart.ejs', {product: product}) %>
5. </div>
6. <% } %>

**9. Dynamic Routes & Advanced Models**

124. Using Query Params

იმ პარამეტრებზე წვდომა, რომლებიც ლინკში **?**-ნიშნის მერე იწერება **(ანუ იგივე query params)** და **&**-ით გამოიყოფა ხოლმე და სევე ქი-ვალიუ ფეერებს წარმოადგენენ, ანუ: localhost:3000/admin/add/id?key=value , შესაძლებელია შემდეგნაირად:

1. exports.getEditProduct = (req, res, next) => {
2. const editMode = req.query.key
3. ...
4. };

**9. Dynamic Routes & Advanced Models**

133. Wrap Up

აქამდე უბრალოდ გავაუმჯობესეთ აპლიკაცია და უფრო მრავალფეროვანი გავხადეთ. ძირითადი პრინციპი იყო ის, რომ მოდელები კლასებში შევქმენით და მის ინსთანსებს ვიყენებდით (როგორც მონაცემთა ბაზები მოიქცევიან სამომავლოდ). ახლა უბრალოდ მათი სიმულაცია იყო ლოქალურად.

**express routing:**

*https://expressjs.com/en/guide/routing.html*

**10. SQL Introduction**

139. Setting Up MySQL

აქამდე გავიარეთ ზოგადი ცოდნა **SQL**და **NoSQL**ტიპის დეითაბაზებზე, რომლებიც ფოტოებაზეა აღწერილი.

ამ ჯერზე გმაოვიყენებთ SQL მონაცემთა ბაზას, კონკრეტულად MySQL (*https://www.mysql.com/*), საიდანც გადავარ download-ში და ბოლოში არსებულ ლინკს -**MySQL Community (GPL) Downloads** - მივყვები, საიდანაც სასურველია ავირჩიო - **MySQL Installer for Windows** - რომელიც ჩაწერს ვებინსტალერს, სადაც უნდა მოვნიშნო აუცილებლად: **MySQL Server** და **MySQL Workbenc**h და გადმოვაწერინო, შემდეგ კი დაინსტალირება როცა დასრულდება, კონფიგურაციის პროცესში უნდა მივუთითო:

* Type and Networking - default
* Authentication Method - Use Legacy...
* Accounts and Roles - passwords & default
* Windows Service - check Configure MySQL Servers...
* Apply Configuration - Execute

მას შემდეგ, რაც დაინსტალირდება, შემიძლია MySQL Workbanch-ზე შევიდე და ჩემს სერვერზე დავლოგინდე, შემდეგ ამ აპლიკაციაში schemas-ში გადავდივარ და ახალ სქემას ვქმნი (node-complete).

**10. SQL Introduction**

140. Connecting our App to the SQL Database

**mysql**-ის ნოუდში გამოსაყენებლად საჭიროა შემდეგი პაკეჯის დაინსტალირება: npm install --save mysql2

შემდეგ კი, ჩვენს შემთხვევაში, ცალკე ფოლდერში ვქმნით**database.js** ფაილს, სადაც ჯერ ვაიმპორტებთ და შემდეგ ცვლადში ვიძახებთ **.createPool()** მეთოდს, რომელიც**.createConnection()**-ისგან იმით განსხვადება, რომ ერთჯერადად არ ამყარებს ქონექშენს და ყოველ ჯერზე ახალი ქიურის გაგზავნა არის შესაძლებელი. იგი მიიღებს არგუმენტად ობიექტს, რომელშიც უნდა გაიწეროს სერვერის კონფიგურაცია. შემდეგ კი დაექსპორტდეს ეს ცვლადი მეთოდით **.promise():**

1. const mysql = require('mysql2');
3. const pool = mysql.createPool({
4. host: 'localhost',
5. user: 'root',
6. database: 'node-complete',
7. password: 'charkviani1616'
8. })
10. module.exports = pool.promise();

**10. SQL Introduction**

140. Connecting our App to the SQL Database

შემდეგ კი აპლიკაციაში ვაიმპორტებთ ამას.

მის გასაშვებად გამოვიყენებთ .execute() ბრძანებას, რომელიც არგუმენტად მიიღებს სიქუალ სინტაქსში დაწერილ სტრინგს (რომელიც ცალკე ენასავითაა):

1. const db = require('./util/database');
2. db.execute('SELECT \* FROM products');

ამისთვის საჭიროა **products**-ის თეიბლი შევქმნათ ჩვენს მიერ შექმნილ სქიმაში.

**10. SQL Introduction**

141. Basic SQL & Creating a Table

**MySQL**-ის მხარეს ბორკბენჩზე უნდა შევქმნათ ახალი თეიბლი, სადაც უნდა მივუთითოთ რა ტიპის დეითას მიიღებს ეს დეითი თავისი ტიპებით და ა.შ.

იქვე შეგვიძლია რაიმე დამი დეითა შევქმნათ, შემდეგ კი მის მისაღებად, ჩვენს ფუნქციას, რამდენადაც**.promise()**-ით დავაექსპორტეთ, შეგვიძლია**.then()** ჩეინი მივაბათ:

1. db.execute('SELECT \* FROM products').then().catch()

**10. SQL Introduction**

142. Retrieving Data

დეითას კი ჩეულებრიც მივიღებთ მასივების სახით:

1. db.execute("SELECT \* FROM products")
2. .then(result => {
3. console.log(result[0])
4. })
5. .catch(err => {
6. console.log(err)
7. });

**10. SQL Introduction**

143. Fetching Products

ახლა კი მოდელში უბრალოდ უნდა  გამოვიძახოთ შემდეგი ბრძანება და ფუნქციას დავაბრუნებინოთ, სხვაგან რომ გვქონდეს **.then()** ჩეინზე წვდომა (დაიმპორტების მერე):

1. static fetchAll() {
2. return db.execute('SELECT \* FROM products');
3. }

შემდეგ კი ასე გამოვიყენებთ:

1. exports.getIndex = (req, res, next) => {
2. Product.fetchAll().then(([rows, fieldData]) => {
3. res.render("shop/index", {
4. prods: rows,
5. pageTitle: "Shop",
6. path: "/",
7. });
8. }).catch(console.log);
9. };

**10. SQL Introduction**

145. Inserting Data Into the Database

მონაცემების შენახვა შემდეგნაირად ხდება:

1. save() {
3. return db.execute('INSERT INTO products (title, price, imageUrl, description) VALUES (?, ?, ?, ?)', [this.title, this.price, this.imageUrl, this.description])
5. }

აქაც SQL სინტაქსით ვწერთ. თანმიმდევრობები უნდა დაემთხვეს ბაზაში არსებული თეიბლის თანმიმდევრობებს.

**10. SQL Introduction**

146. Fetching a Single Product with the "where" Condition

ხოლო როცა მხოლოდ ერთი დეითას მოტანა გვსურს, შემდეგნაირად უნდა დავწეროთ:

1. static findById(id) {
2. return db.execute('SELECT \* FROM products WHERE products.id = ?', [id])
3. }

shop.js:

1. exports.getProduct = (req, res, next) => {
2. const prodId = req.params.productId;
3. Product.findById(prodId).then(([product]) => {
4. res.render("shop/product-detail", {
5. product: product[0],
6. pageTitle: product.title,
7. path: "/products",
8. });
9. }).catch(console.log)
10. };

**11. Understanding Sequelize**

151. Connecting to the Database

**Sequelize**- გვეხმარება იმაში, რომ სიქუალ კოდის წერის ნაცვლად, ჯავასკრიპტის ობიექტები შევქმნათ, რომლებიც სცენის მიღმა იგივეს გააკეთებენ, რასაც ჩვენ სიქუალ კოდის წერისას ვაკეთებდით.

*(ფოტოებზეცაა ახსნილი).*

იგი ჩვეულებრივი პაკეჯია, რომელსაც ვაინსტალირებთ შემდეგი ბრძანებით:

npm install --save sequelize

მას მერე, თუ **MySQL**-ში რაიმე თეიბლი გვაქვს, უნდა დავდროპოთ და შემდეგ წინა **database.js**-ის კონტენტს შემდეგით ჩავანაცვლებთ:

1. const Sequelize = require('sequelize');
3. const sequelize = new Sequelize('node-complete', 'root', 'charkviani1616', {dialect: 'mysql', host: 'localhost'})

6. module.exports = sequelize;

**11. Understanding Sequelize**

152. Defining a Model

ახლა უკვე მოდელების დადეკლარირების დროა. ძველ მოდელს ვშლით, რადგან ახლა უკვე სიქუალ დეითაბეიზის მოდელს გამოვიყენებთ ჩვენის ნაცვლად.

აქ დაგვჭირდება როგორც ძირითად მოდული **sequelize**-ს, ასევე იმპორტი **utils/database.js**-იდან, სადაც დეითაბაზასთან კავშირს ვამყარებთ.

ამ ორ ცვლადზე დაყრდნობით კი ვქმნით ახალ მოდელებს:

1. const Sequelize = require('sequelize');
3. const sequelize = require('../util/database');
5. const Product = sequelize.define('product', {
6. id: {
7. type: Sequelize.INTEGER,
8. autoIncrement: true,
9. allowNull: false,
10. primaryKey: true
11. },
12. title: Sequelize.STRING,
13. price: {
14. type: Sequelize.DOUBLE,
15. allowNull: false
16. },
17. imageUrl: {
18. type: Sequelize.STRING,
19. allowNull: false
20. },
21. description: {
22. type: Sequelize.STRING,
23. allowNull: false
24. }
25. })
27. module.exports = Product

**11. Understanding Sequelize**

153. Syncing JS Definitions to the Database

იმისათვის, რომ აპლიკაციის დაწყებისას, ეგრევე მოხდეს ჩვენი მოდელებისთვის თეიბლების შექმნა, ***app.js***-ის დონეზე უნდა დავაიმპორტოთ **util/database.js**-ის დეითაბეიზთან დაქონექთება და შემდეგ, **app.listen()**-თან გამოვიძახოთ დაიმპორტების ცვლადი და მივაბათ მეთოდი **.sync(),** რომელიც ზემოთ ხსენებულს გააკეთებს:

1. const sequelize = require('./util/database');
3. sequelize.sync()

ასევე მასშიც შეგვიძლია ჩავსვათ სერვერის დაწყების ბრძანება:

1. sequelize.sync().then((result)=>{
2. console.log(result)
3. app.listen(3000, console.log("Server is running"));
4. }).catch(console.log)

**Sequelize Library:**

*https://sequelize.org/*

**11. Understanding Sequelize**

154. Inserting Data & Creating a Product

რამდენადაც, მოდელებში უკვე შევქმენით ჩვენი პროდუქტის მოდელი და მას ვაექსპორტებთ, მისი ინტანსების გამოყენება ყველგან შეგვიძლია, სადაც დავაიმპორტებთ (მასში არსებული მეთოდების).

მაგალითად, რაიმეს დამატებისთვის:

1. const Product = require("../models/product");
3. exports.postAddProduct = (req, res, next) => {
4. const title = req.body.title;
5. const imageUrl = req.body.imageUrl;
6. const price = req.body.price;
7. const description = req.body.description;
9. Product.create({
10. title: title,
11. price: price,
12. imageUrl: imageUrl,
13. description: description
14. }).then(()=>{
15. console.log('Created product!')
16. }).catch(console.log)
18. };

**11. Understanding Sequelize**

156. Retrieving Data & Finding Products

ასე მოგვაქვს ყველაფერი:

1. Product.findAll().then(products => {
2. res.render("shop/product-list", {
3. prods: products,
4. pageTitle: "All Products",
5. path: "/products",
6. });
7. }).catch(console.log)

**11. Understanding Sequelize**

157. Getting a Single Product with the "where" Condition

ერთი კონკრეტული პროდუქტის მოტანის ორ გზას ვახსენებ აქ, ერთია წინა მეთდით, უბრალოდ არგუმენტად უნდა განვუსაზღვროთ რომელი როუ წამოიღოს, სადაც აიდის ჩავაწვდით, მეორე კი უბრალოდ აიდის იღებს არგუმენტად და ერთ კონკრეტულ როუს ესე აბრუნებს:

ერთი მეთოდი:

1. Product.findAll({where: {id: prodId}}).then(products => {
2. res.render("shop/product-detail", {
3. product: products[0],
4. pageTitle: products[0].title,
5. path: "/products",
6. });
7. }).catch(console.log)

მეორე მეთოდი:

1. Product.findByPk(prodId).then((product) => {
2. res.render("shop/product-detail", {
3. product: product,
4. pageTitle: product.title,
5. path: "/products",
6. });
7. }).catch(console.log)

**11. Understanding Sequelize**

159. Updating Products

აფდეითისთვის ვიყენებთ **findByPk()**-ს მიერ დაბრუნებულ ცვლადში არსებულ მეთოდს**.save()**რომელიც თუ არსებობს დააფდეითებს კონკრეტულ პროდუქტს, თუ არა, შექმნის:

1. exports.postEditProduct = (req, res, next) => {
2. const prodId = req.body.productId;
3. const updatedTitle = req.body.title;
4. const updatedPrice = req.body.price;
5. const updatedImageUrl = req.body.imageUrl;
6. const updatedDesc = req.body.description;
8. Product.findByPk(prodId)
9. .then(product => {
10. product.title = updatedTitle;
11. product.price = updatedPrice;
12. product.description = updatedDesc;
13. product.imageUrl = updatedImageUrl;
14. return product.save();
15. })
16. .then(() => {
17. console.log("UPDATED PRODUCT!");
18. res.redirect("/admin/products");
19. })
20. .catch(console.log);
21. };

**11. Understanding Sequelize**

160. Deleting Products

რაიმეს წაშლისას შეგვიძლია გამოვიყენოთ **destroy()** მეთოდი პირდაპირ ინსტანსზე, თუმცა ამ შემთხვევაში მას უნდა ჩავაწოდოთ ობიექტი და გავუწეროთ პარამეტრები იმის, რაც წაშალოს, არ უბრალოდ ჯერ ვიპოვნოთ და შემდეგ ნაპოვნს უბალოდ მივაბათ ეს მეთოდი და იგი წაიშლება:

1. exports.postDeleteProduct = (req, res, next) => {
2. const prodId = req.body.productId;
3. Product.destroy({});
4. Product.findByPk(prodId).then(product => {
5. return product.destroy()
6. })
7. .then(() => {
8. console.log('DESTROYED PRODUCT!')
9. res.redirect("/admin/products");
10. })
11. .catch(console.log)
12. };

**11. Understanding Sequelize**

162. Adding a One-To-Many Relationship

ასოციაციები საჭიროა იმისათვის, რომ სხვადასხვა თეიბლები დავაკავშიროთ ერთმანეთთან. მაგალითად, მომხმარებელი და პროდუქტი.

ამისთვის ჯერ ორივეს მოდელი უნდა გვქონდეს შექმნილი, რათა ორივეს თეიბლი არსეობობდეს.

შემდეგ კი**app.js**-ის დონეზე, **async()**-მდე უნდა გაიწეროს მათი კავშირები და ასევე დაიმპორტდნენ მოდელები:

1. const Product = require('./models/product');
2. const User = require('./models/user');
4. Product.belongsTo(User, {constraints: true, onDelete: 'CASCADE'});

აქ **onDelete**ნიშნავს, თუ მომხმარებელი წაიშლება, ყველა მასზე მიმაგრებული პროდუქტი წაიშალოს (ანუ მის მიერ შექმნილი).

1. User.hasMany(Product);

ხოლო ბოლოს კი, **async()** მეთოდში უნდა ჩავაწოდოთ ობიექტი არგუმენტად, რომელიც არსებულ დეითაბეიზის კონფიგურაციას ახალს გადააწერს (ფროდაქშენში არაა რეკომენდირებული):

1. sequelize.sync({force: true}).then(()=>{
2. app.listen(3000, console.log("Server is running"));
3. }).catch(console.log)

**11. Understanding Sequelize**

163. Creating & Managing a Dummy User

რამდენადაც ავტორიზაცია არ გვაქვს, შეგვიძლია მაინც ხელოვნურად შევქმნათ მომხმარებელი**.async()** მეთოდშივე, რომელიც იქმნება ინსტანსზე მიბმული ბრძანებით **.create()**.

ასევე შეგვიძლია მიდლვეარი შევქმნათ, სადაც ყველა რექვესტს მივაბამთ ჩვენს იუზერს, რომელსაც შემდეგ ყველა რექვესტის ჰენდლინგისას მივწვდებით.

ასინქრონში მომხმარებლის შექმნა:

1. sequelize
2. .sync()
3. .then(() => {
4. return User.findByPk(1);
5. })
6. .then(user => {
7. if(!user){
8. return User.create({name: 'Gio', email: 'gi.charkviani@gmail.com'})
9. }
10. return user
11. })
12. .then(user => {
13. app.listen(3000, console.log("Server is running"));
14. })
15. .catch(console.log);

მიდლვეარი:

1. app.use((req, res, next) => {
2. User.findByPk(1).then(user => {
3. req.user = user
4. next()
5. }).catch(console.log)
6. })

**11. Understanding Sequelize**

164. Using Magic Association Methods

ამის მერე ყველგან შეგვიძლია უბრალოდ პროდუქტს ჩავამატოთ ახალი ფროფერთი **userId**მაგალითად, რომელიც ავტომატურად შეიქმნა ამ ორი თეიბლის დაკავშირებისას (რომელიც მიდლვეარით მივაბით რექვესტს):

1. Product.create({
2. title: title,
3. price: price,
4. imageUrl: imageUrl,
5. description: description,
6. userId: req.user.id
7. })

თუმცა ამის უკეთეს ალტერნატივად შეიძლება გამოვიყენოთ თავად user-ში ჩაშენებული მეთოდი **createProduct()**, რომელიც გაჩნდა მას შემდეგ, რაც ასოციაცია დავამყარეთ ორ თეიბლს შორის:

1. req.user.createProduct({
2. title: title,
3. price: price,
4. imageUrl: imageUrl,
5. description: description
6. })

**11. Understanding Sequelize**

165. Fetching Related Products

ამის შემდეგ კი შეგვიძლია პირდაპირ იუზერზე მივაბათ მეთოდები, რათა კონკრეტულ იუზერზე მიბმული დეითა წამოიღოს, მაგ:

1. req.user.getProducts()

**11. Understanding Sequelize**

166. One-To-Many & Many-To-Many Relations

ასევე, შეგვიძლია უფრო მრავალფეროვანი კავშირები დავამყაროთ თეიბლებს შორის, მაგალითად ჩვენს შემთხვევაში ქართი, იგივე კალათა, მხოლოდ ერთი შეიძლება ერთ მომხმარებელს ეკუთვნოდეს, ასევე იგი შეიძლება ბევრ პროდუქტს ჰქონდეს და ასევე ბევრ პროდუქტს შეიძლება ჰქონდეს იგი, ესეც ასევე **app.js**-ის დონეზე იწერება (საჭირო იმპორტებით):

1. Product.belongsTo(User, { constraints: true, onDelete: "CASCADE" });
2. User.hasMany(Product);
3. User.hasOne(Cart);
4. Cart.belongsTo(User);
5. Cart.belongsToMany(Product, {through: CartItem});
6. Product.belongsToMany(Cart, {through: CartItem});

throught: CartItem - ადეკლარირებს, თუ რომელი კონკრეტული მოდული დააკავშირებს მათ, ანუ გამოიყენებენ.

**11. Understanding Sequelize**

167. Creating & Fetching a Cart

ასევე საჭირო იქნება**async()** ჩეინში როგორც მომხამრებელი შევქმენით, ასევე შევქმნათ **cart**-ი, რათა შემდეგ **user**-ში მასზე წვდომა გვქონდეს:

1. .then(user => {
2. return user.createCart();
3. })
4. .then(cart => {
5. app.listen(3000, console.log("Server is running"));
6. })

*გაგრძელდება...*

**11. Understanding Sequelize**

167. Creating & Fetching a Cart

ამის მერე კი, სადაც **user**-ს ვწვდებით მიდლვეარის წყალობით, ასევე მასში იარსებებს მეთოდი **.getCart()**, რომელიც დააბრუნებს ისეთ არგუმენტს, რომელსაც ექნება მეთოდი **.getProducts()**, ეს კი მოგვცემს წვდომას ყველა კალათაში არსებულ პროდუქტზე:

1. exports.getCart = (req, res, next) => {
2. req.user
3. .getCart()
4. .then((cart) => {
5. cart
6. .getProducts()
7. .then((products) => {
8. res.render("shop/cart", {
9. path: "/cart",
10. pageTitle: "Your Cart",
11. products: products,
12. });
13. })
14. .catch(console.log);
15. })
16. .catch(console.log);
17. };

ეს ყველაფერი კი საწყისში დეკლარირებული კავშირების გამო ხდება...

**11. Understanding Sequelize**

169. Adding Existing Products & Retrieving Cart Items

დეითას დასამატებლად ასე მრავალფეროვნად დაკავშირებულ თეიბლებში შემდეგი ტიპის სინტაქსს გამოვიყენებთ:

1. exports.postCart = (req, res, next) => {
2. const prodId = req.body.productId;
3. let fetchedCart;
4. let newQuantity = 1;
5. req.user.getCart()
6. .then(cart => {
7. fetchedCart = cart;
8. return cart.getProducts({where: {id: prodId}})
9. })
10. .then(products => {
11. let product;
12. if(products.length > 0){
13. product = products[0]
14. }
16. if(product){
17. const oldQuantity = product.cartItem.quantity;
18. newQuantity = oldQuantity + 1;
19. return product;
20. }
21. return Product.findByPk(prodId);
22. })
23. .then(product => {
24. return fetchedCart.addProduct(product, {through: { quantity: newQuantity } })
25. })
26. .then(() => {
27. res.redirect('/cart')
28. })
29. .catch(console.log)
30. };

**11. Understanding Sequelize**

169. Adding Existing Products & Retrieving Cart Items

ასევე მცირე ეჯასტმენტები **EJS**-შიც, რამდენადაც სხვანაირი ტიპის დეითას ვწვდებით:

1. <p><%= p.title %> (<%= p.cartItem.quantity %>)</p>

**11. Understanding Sequelize**

170. Deleting Related Items & Deleting Cart Products

წაშლისას კი:

1. exports.postCartDeleteProduct = (req, res, next) => {
2. const prodId = req.body.productId;
3. req.user
4. .getCart()
5. .then((cart) => {
6. return cart.getProducts({ where: { id: prodId } });
7. })
8. .then((products) => {
9. const product = products[0];
10. return product.cartItem.destroy();
11. })
12. .then(() => {
13. res.redirect("/cart");
14. })
15. .catch(console.log);
16. };

**11. Understanding Sequelize**

173. Resetting the Cart & Fetching and Outputting Orders

ამ სექციაში ახალი მოდელი შევქმენით და შესაბამისად, დავაკავშირეთ სხვა თეიბლებთან. არის გარკვეული სიახლეები, რაც კოდშიც წერია.**მოკლედ კი:**

1. .then((products) => {
2. return req.user
3. .createOrder()
4. .then((order) => {
5. return order.addProducts(
6. products.map((product) => {
7. product.orderItem = { quantity: product.cartItem.quantity };
8. return product;
9. })
10. );
11. })
12. .catch(console.log);

**12. Working with NoSQL & Using MongoDB**

180. Installing the MongoDB Driver

*აქამდე რაც იყო, თეორიები - ყველაფერი ფოტოებზეა აღწერილი, დანარჩენი ყველაფერი იგივეა, რაც უკვე ვიცი, ანუ რაც****MongoDB****-ს შეეხება.*

მისი დრაივერის დასაყენებლად ვუშვებთ ბრძანებას:

npm install --save mongodb

შემდეგ კი**app.js**-დან ყველაფერი წავშალო, რაც **MySQL**-ს უკავშირდება.

შემდეგ კი ასევე **util**-იდან **database.js**-ფალიში ვშლით ყველაფერს და ვადეკლარირებთ უკე ახალ და სამარადჟამო სამუშაო დეითაბეიზს - **MONGODIIIBIII**-ს:

1. const mongodb = require("mongodb");
2. const MongoClient = mongodb.MongoClient;
4. const mongoConnect = (callback) => {
5. MongoClient.connect(
6. "mongodb+srv://giorgi:charkviani1616@cluster0.bobiq.mongodb.net/myFirstDatabase?retryWrites=true&w=majority"
7. )
8. .then(client => {
9. console.log("Connected");
10. callback(client)
11. })
12. .catch(console.log);
13. };
15. module.exports = mongoConnect;

**12. Working with NoSQL & Using MongoDB**

180. Installing the MongoDB Driver

შემდეგ კი მას **app.js**-ში დავაიმპორტებთ და რამდენადაც ფუნქციას ვაექსპორტებთ, იქ უბრალოდ გამოვიძახებთ და მის ქოლბაქს მოვიხმართ:

1. const mongoConnect = require('./util/database');
3. mongoConnect(client => {
4. console.log(client)
5. app.listen(3000, console.log("Server is running"));
6. })

**12. Working with NoSQL & Using MongoDB**

182. Finishing the Database Connection

ამ ლექციაში დეითაბეიზის გამოძახების პრინციპი შევცვალეთ. **cleint**-ზე მიბმულ მეთოდს **.db()**-ს გამოვიყენებთ ნაცვლად იმისა, რომ ქოლბაქში ჩაგვეწოდებინა client-ი. ასევე შესაძლებელია **.db()**-ის არგუმენტად გადავცეთ რაიმე სტრინგი, რაც დეითაბეიზის სახელიც იქნებოდა.

**database.js:**

1. const mongodb = require("mongodb");
2. const MongoClient = mongodb.MongoClient;
4. let \_db;
6. const mongoConnect = (callback) => {
7. MongoClient.connect(
8. "mongodb+srv://giorgi:charkviani1616@cluster0.bobiq.mongodb.net/shop?retryWrites=true&w=majority"
9. )
10. .then(client => {
11. console.log("Connected");
12. \_db = client.db()
13. callback()
14. })
15. .catch(err => {
16. console.log(err);
17. throw err;
18. });
19. };
21. const getDb = () => {
22. if(\_db) {
23. return \_db
24. }
25. throw 'No database found!'
26. }
28. exports.mongoConnect = mongoConnect;
29. exports.getDb = getDb;

**12. Working with NoSQL & Using MongoDB**

183. Using the Database Connection

**app.js**-შიც უნდა დავაქონექთოდ, რამდენადაც ცენტრალური დეითაბეიზიდან დაქონექთების ობიექტს ვაექსპორტებთ:

1. const mongoConnect = require('./util/database').mongoConnect;
3. mongoConnect(() => {
4. app.listen(3000, console.log("Server is running"));
5. })

ხოლო, ჩვენი მოდელი შემდეგნაირად გამოიყურება **(product.js):**

1. const getDb = require('../util/database').getDb;
3. class Product {
4. constructor(title, price, description, imageUrl){
5. this.title = title;
6. this.price = price;
7. this.description = description;
8. this.imageUrl = imageUrl;
9. }
11. save() {
12. const db = getDb();
13. db.collection('products').insertOne(this)
14. .then(result => {
15. console.log(result)
16. })
17. .catch(console.log)
18. }
19. }

22. module.exports = Product;

**12. Working with NoSQL & Using MongoDB**

184. Creating Products

**product.js**-ში, სადაც **getDb**შემოგვაქვს, რამდენადაც იგი წარმოადგენს ჩვენს დეითაბეიზსთან ქონექშენის ინსტანსს, შეგვიძლია მასზე მივაბათ ისეთი მეთოდი, როგორიცაა **.collection()**, რომელიც არგუმენტად იღებს კოლექციის სახელს და შემდგომ ამ ფუნქციაზე მივაბათ ყველა ის მოქმედებები, რომლებსაც ამ კონკრეტულ კოლექციაში განვახორციელებთ, მაგალთად **.insertOne()**, რომელიც ობიექტს იღებს არგუმენტად.

მაგ.:

1. const db = getDb();
2. db.collection('products').insertOne(this)
3. .then(result => {
4. console.log(result)
5. })
6. .catch(console.log)

**12. Working with NoSQL & Using MongoDB**

184. Creating Products

შემდეგ კი, კონტროლერში, ჩვეულებრივ ვაიმპორტებთ თავად მოდელს (როგორც აქამდე) და მას ვიძახებთ და ვქმნით მის ახალ ინსტანსს.

**admin.js:**

1. const Product = require("../models/product");
3. exports.postAddProduct = (req, res, next) => {
4. const title = req.body.title;
5. const imageUrl = req.body.imageUrl;
6. const price = req.body.price;
7. const description = req.body.description;
9. const product = new Product(title, price, description, imageUrl);
11. product
12. .save()
13. .then(() => {
14. console.log("Created product!");
15. res.redirect("/admin/products");
16. })
17. .catch(console.log);
18. };

**12. Working with NoSQL & Using MongoDB**

185. Understanding the MongoDB Compass

ამ ლექციაში **MongoDB Compass**-ის გამართვა მიმდინარეობს...

**12. Working with NoSQL & Using MongoDB**

186. Fetching All Products

**MongoDB**-დან მონაცემების მისაღებად, ისევ**.collection()**-ს უნდა მივაბათ მეთოდი**.find()**, რომლის კონფიგურირებაც შეიძლება, მაგალთად ობიექტის ჩაწოდებით და მითითებით, თუ რომელი ფროფერთის მქონე დეითა მოიტანოს.

ასევე **.find()** ასევე შეუძლია დააბრუნოს მთლიანი დეითა, რაც შეიძლება სასურველი არ იყოს დიდი დეითას შემთხვევაში.

მას ასევე ებმება მეთოდი**.toArray()**, რაც მოტანილ დეითას მასივად გადააკეთებს, რომელიც ფრომისს აბრუნებს.

**product.js:**

1. static fetchAll() {
2. const db = getDb();
3. return db
4. .collection("products")
5. .find()
6. .toArray()
7. .then((products) => {
8. console.log(products);
9. return products;
10. })
11. .catch(console.log);
12. }

**12. Working with NoSQL & Using MongoDB**

186. Fetching All Products

დეითას მიღებისას კი, უბრალოდ:

**shop.js:**

1. const Product = require("../models/product");
3. Product.fetchAll()
4. .then((products) => {
5. res.render("shop/product-list", {
6. prods: products,
7. pageTitle: "All Products",
8. path: "/products",
9. });
10. })
11. .catch(console.log);

**12. Working with NoSQL & Using MongoDB**

187. Fetching a Single Product

ერთი კონკრეტული დეითას მოსაძიებლად, ასევე შეგვიძლია **.find()** მეთოდი გამოვიყენოთ, არგუმენტად კი ჩავაწოდოთ ფროფერთი და აიდი, რომელ დეითასაც ვეძებთ, შემდეგ კი მივაბათ **.next()** მეთოდი, რომელიც უკანასკნელ დეითას გამოიტანს:

**product.js:**

1. const mongodb = require('mongodb');
2. ...
3. static findById(prodId) {
4. const db = getDb();
5. return db
6. .collection("products")
7. .find({ \_id: new mongodb.ObjectId(prodId)})
8. .next()
9. .then((product) => {
10. console.log(product)
11. return product
12. })
13. .catch(console.log);
14. }

**mongodb.ObjectId()** - მის გარეშე მონგო ვერ მიხვდება, რომ აიდით ვეძებთ, რადგან იგი წარმოადგენს სპეციალურ ტიპს, რომელსაც BISON-ნის საშალებით ცნობს.

*გაგრძელდება...*

**12. Working with NoSQL & Using MongoDB**

187. Fetching a Single Product

ასევე მისი გამოყენებისას:

**shop.js:**

1. const prodId = req.params.productId;
2. Product.findById(prodId)
3. .then((product) => {
4. res.render("shop/product-detail", {
5. product: product,
6. pageTitle: product.title,
7. path: "/products",
8. });
9. })
10. .catch(console.log);

ასევე გასათვალისწინებელია, რომ თუ სადმეს **id**-ით ვიძახებდით, **\_id**-ით ჩავანაცვლოთ.

**12. Working with NoSQL & Using MongoDB**

190. Finishing the "Update Product" Code

დასააფდეითებლად სასურველია გამოვიყენოთ **.updateOne()**მეთოდი, რომელიც პირველ არგუმენტად იღებს საძიებო ფროფერთის, მაგალითად **\_id**, ხოლო მეორე არგუმენტად ობიექტს, რომელსაც აქვს ფროფერთ**$set**: და მის მნიშვნელობად შეგვიძლია უკვე ჩავაწოდოთ ის მონაცემები, რასაც ვანახლებთ კონკრეტულ დეითაში:

1. dbOp = db.collection('products').updateOne({\_id: new mongodb.ObjectId(this.\_id)}, {$set: this})

შემდეგ კი უბრალოდ კონკრეტული კონტროლერიდან გამოვიძახოთ და დავააფდეითოთ:

1. const mongodb = require("mongodb");
2. const ObjectId = mongodb.ObjectId;
4. ...
5. const product = new Product(
6. updatedTitle,
7. updatedPrice,
8. updatedDesc,
9. updatedImageUrl,
10. new ObjectId(prodId)
11. );
13. product.save()
14. .then(() => {
15. console.log("UPDATED PRODUCT!");
16. res.redirect("/admin/products");
17. })
18. ...

**12. Working with NoSQL & Using MongoDB**

192. Deleting Products

წაშლისას კი ვიყენებთ მეთოდს**.deleteOne()**, რომელიც კონფიგურაციის ობიექტს იღებს არგუმენტად, ანუ რითი მონახოს დეითა, შემდეგ კი ამას გამოვიყენებთ ყველგან:

1. static deleteById(prodId) {
2. const db = getDb();
3. return db.collection('products').deleteOne({\_id: new mongodb.ObjectId(prodId)})
4. .then(() => {
5. console.log('Deleted!')
6. })
7. .catch(console.log)
8. }

**12. Working with NoSQL & Using MongoDB**

195. Storing the User in our Database

ამ ლექციებში მომხამრებლის მოდელი შევქმენით, რომელსაც ასევე მიდლვეარსაც ვაწვდით **req**-ში, დაასევე პროდუქტის მოდელსაც ვაბამთ მომხმარებლის **id**-ის.

**user.model:**

1. class User {
2. constructor(username, email) {
3. this.name = username;
4. this.email = email;
5. }
7. save() {
8. const db = getDb();
9. return db.collection('users').insertOne(this);
10. }
12. static findById(userId) {
13. const db = getDb();
14. return db
15. .collection('users')
16. .findOne({\_id: new ObjectId(userId)})
17. .then(user => {
18. console.log(user)
19. return user
20. })
21. .catch(console.log)
22. }
23. }

**12. Working with NoSQL & Using MongoDB**

199. Displaying the Cart Items

მაგალითად, თუ გვსურს **MongoDB**-დან მივიღოთ ის კონკრეტული დეითა, რომლებიც ატარებენ გარკვეულ **ID**-ებს, **$in** პარამეტრს ვიყენებთ:

1. getCart() {
2. const db = getDb();
3. const productIds = this.cart.items.map((i) => {
4. return i.productId;
5. });
6. return db
7. .collection("products")
8. .find({ \_id: { $in: productIds } })
9. .toArray()
10. .then((products) => {
11. return products.map((p) => {
12. return {
13. ...p,
14. quantity: this.cart.items.find((i) => {
15. return i.productId.toString() === p.\_id.toString();
16. }).quantity
17. };
18. });
19. })
20. .catch(console.log);
21. }

**12. Working with NoSQL & Using MongoDB**

199. Displaying the Cart Items

შემდეგ კი უბრალოდ მივიღოთ დეითა

**shop.js:**

1. exports.getCart = (req, res, next) => {
2. req.user
3. .getCart()
4. .then((products) => {
5. res.render("shop/cart", {
6. path: "/cart",
7. pageTitle: "Your Cart",
8. products: products,
9. });
10. })
11. .catch(console.log);
12. };

**12. Working with NoSQL & Using MongoDB**

201. Deleting Cart Items

წაშლას კი შემდეგნაირად ვიზამთ:

1. deleteItemFromCart(productId) {
2. const updatedCartItems = this.cart.items.filter(item => {
3. return item.productId.toString() !== productId.toString()
4. })
5. const db = getDb();
6. return db
7. .collection("users")
8. .updateOne(
9. { \_id: new ObjectId(this.\_id) },
10. { $set: { cart: {items: updatedCartItems} } }
11. );
12. }

**12. Working with NoSQL & Using MongoDB**

204. Getting Orders

თუ მაგალითად ჩანესტილ დეითას უნდა მივწვდეთ და ისე გავფილტროდ, შემდეგნაირად უნდა დავწეროთ:

1. getOrders() {
2. const db = getDb();
3. return db
4. .collection("orders")
5. .find({ "user.\_id": new ObjectId(this.\_id) })
6. .toArray();
7. }

**13. Working with Mongoose**

211. Connecting to the MongoDB Server with Mongoose

**Mongoose**-ზე ახსნა ფოტოებზეა ასახული...

***Library****: https://mongoosejs.com/*

**Mongoose**-ის დასაინსტალირებლად ვუშვებთ ბრძანებას: npm install --save mongoose

შემდეგ კი უბრალოდ დავაიმპორეტებთ მონგუსს და მასში გამოვიძახებთ მეთოდს**.connect()**, რომელიც არგუმენტად დაქონექთების ლინკს მიიღებს და სულ ესაა (იგი ასევე აბრუნებს ფრომისს):

1. const mongoose = require("mongoose");
3. mongoose
4. .connect(
5. "..."
6. )
7. .then(() => {
8. app.listen(3000, () => {
9. console.log("connected");
10. });
11. })
12. .catch(console.log);

**13. Working with Mongoose**

212. Creating the Product Schema

მონგუსში ვიყენებ ე.წ. სქიმას (Schema), რომელიც გვეხმარება დეითას ტიპის შექმნაში, ანუ მისი მოდელის განსაზღვრაში.

ამისთვის, ჯერ ცალკე ფაილში ვაიმპორტებთ მონგუსს, შემდეგ კი მანდედან ვიძახებთ Schema-ს, შემდეგ კი, Schema-დან უკვე გამომდინარეობს ახალი ობიექტი, შესაბამისად new ქივორდით შევქმნით ახალ სკიმას, რომელსაც არგუმენტად ჩაეწოდება ობიექტი და შიგნით დეკლარირდება კონკრეტული კოლექციის მოდელი, თავისი ჩაშენებული პარამეტრებით, მაგალითად required: true:

1. const mongoose = require('mongoose');
3. const Schema = mongoose.Schema;
5. const productSchema = new Schema({
6. title: {
7. type: String,
8. required: true
9. },
10. price: {
11. type: Number,
12. required: true
13. },
14. description: {
15. type: String,
16. required: true
17. },
18. imageUrl: {
19. type: String,
20. required: true
21. }
22. });

**13. Working with Mongoose**

213. Saving Data Through Mongoose

შემდეგ კი ამ ფაილიდან დავაექსპორტებთ **mongoose**-ის**.model()**-ს. იგი იმ ბლუპრინტს, ანუ სქემას, მოდელად აქცევს. პირველ არგუმენტად კოლექციის სახელს იღებს, მეორედ კი - თავად სქემას:

1. module.exports = mongoose.model('Product', productSchema)

ამის მერე, კონკრეტულ კონტროლერში გადავალთ და დაიმპორტებული მოდელიდან შევქმნით ახალ ინსტანსს კონკრეტული მოდელის, რომელსაც ობიექტს ჩავაწვდით არგუმენტად. რომელიც მოდელის სქემის სტრუქტურას მიჰყვება, შემდეგ კი მას თავად აქვს **.save()** მეთოდი, რომელიც წინაზე ჩვენს მიერ იყო შექმნილი. ეს მეთოდი **mongoose**-იდან მოდის. იგი ტექნიკურად ფრომისს არ აბრუნებს, თუმცა მონგუსი მაინც გვაძლევს მსგავსი ფორმის მეთოდებს, რომლებიც შეგვიძლია მივაბათ.

*კოდი შემდეგში...*

**13. Working with Mongoose**

213. Saving Data Through Mongoose

დეითას დამატება

**admin.js (controller):**

1. exports.postAddProduct = (req, res, next) => {
2. const title = req.body.title;
3. const imageUrl = req.body.imageUrl;
4. const price = req.body.price;
5. const description = req.body.description;
7. const product = new Product({
8. title: title,
9. price: price,
10. description: description,
11. imageUrl: imageUrl,
12. });
14. product
15. .save()
16. .then(() => {
17. console.log("Created product!");
18. res.redirect("/admin/products");
19. })
20. .catch(console.log);
21. };

**13. Working with Mongoose**

214. Fetching All Products

დეითას მიღებისას, სტანდარტული **find()**-ისგან განსხვავებით, მონგუსის **find()** არ აბრუნებს კურსოსრს (თუმცა შეგვეძლო მასზე**.cursor()** მეთოდის დამატებით), ამის ნაცვლად იგი ყველა პროდუქტს აბრუნებს.

1. Product.find()

**13. Working with Mongoose**

215. Fetching a Single Product

მონგუსს ასევე ააქვს მეთოდი **findById()**, რომელიც არგუმენტად მხოლოდ სტრინგს იღებს და თვითონვე აკონვერტირებს მასს ობიექტში და ასე ეძებს კონკრეტულ დეითას:

1. Product.findById(prodId)

**13. Working with Mongoose**

216. Updating Products

ასევე, როდესაც **.findById()**-ის ვიყენებთ, მის მიერ დაბრუნებულ ინფოს ასევე მოჰყვება საჭირო მეთოდებიც, მათ შორისაა **.save()**, ამის წყალობით კი კონკრეტულ დეითას შემდეგნაირად განვაახლებდით:

1. exports.postEditProduct = (req, res, next) => {
2. const prodId = req.body.productId;
3. const updatedTitle = req.body.title;
4. const updatedPrice = req.body.price;
5. const updatedImageUrl = req.body.imageUrl;
6. const updatedDesc = req.body.description;
8. Product.findById(prodId)
9. .then((product) => {
10. product.title = updatedTitle;
11. product.price = updatedPrice;
12. product.description = updatedDesc;
13. product.imageUrl = updatedImageUrl;
14. return product.save();
15. })
16. .then(() => {
17. console.log("UPDATED PRODUCT!");
18. res.redirect("/admin/products");
19. })
20. .catch(console.log);
21. };

**13. Working with Mongoose**

217. Deleting Products

წაშლისას კი მონგუსი შემდეგ მეთოდს გვთავაზობს:

1. Product.findByIdAndRemove(prodId)

**13. Working with Mongoose**

218. Adding and Using a User Model

მომხმარებლის მოდელის შექმნისას გამოვყიენეთ ისეთი ტიპის ფროფერთი, რომელიც ასევე ინახავს სხვა ფროფერთიებს. განსხვავებული ის იყო, რომ ჩვენ შეგვიძლია **Schema**-შ მივწვდეთ **Types**და აქედან დავუდეკლარიროთ **ObjectId**-ის ტიპი, რამდენადაც მას მოელის კონკრეტული ფროფერთი:

1. cart: {
2. items: [
3. {
4. productId: { type: Schema.Types.ObjectId, required: true },
5. quantity: { type: Number, required: true },
6. },
7. ],
8. },

შემდეგ კი იმპორტაციის მერე ჩვეულებრივ, ახალი ინსტანსის შექმნის პრინციპით გამოვიყენებთ, სადაც **.findOne()**-ს თუ არგუმენტს არ ჩავაწვდით, პირველივე დეითას მოიტანს, რასაც ნახავს:

1. User.findOne().then((user) => {
2. if (!user) {
3. const user = new User({
4. name: "Gio",
5. email: "gi.charkviani@gmail.com",
6. cart: {
7. items: [],
8. },
9. });
10. user.save();
11. }
12. });

**13. Working with Mongoose**

218. Adding and Using a User Model

შემდეგ კი მიდლვეარშიც თავისუფლად გამოვიყენებ. ასევე გასათვალისწინებელია, რომ მონგუსი მის ფუნქციებში აბრუნებს ისეთ დეითას, რომელსაც ყველა საჭირო მეთოდი მოჰყვება, აქედან გამომდინარე, ამ დეკლარირების მერე ყველა მეთოდზე მექნება წვდომა:

1. app.use((req, res, next) => {
2. User.findById("619bd8f98fc91fd080263549")
3. .then((user) => {
4. req.user = user;
5. next();
6. })
7. .catch(console.log);
8. });

**13. Working with Mongoose**

219. Using Relations in Mongoose

თუ გვსურს, რომ კავშირი დავამყაროთ ორ მოდელს შორის, მას ფროფერთიდ უნდა გავუწეროთ **ref: 'მოდელი',** ანუ რომელ მოდელთანაც ვაკავშირებთ. ჩვენს შემთხვევაში:

1. userId: {
2. type: Schema.Types.ObjectId,
3. ref: 'User',
4. required: true
5. }

ასევე იგივე გაეწერება საჭირო ადგილზე მეორე მოდელსაც თუ გვსურს.

შემდეგ კი დეითას დამატებისას, რამდენადაც ახალი ფროფერთი აქვს კონკრეტული დეითას მოდელს, უნდა მივუთითოთ მომხმარებლის აიდი, ამის ორი გზაა:**userId: req.user** ან **userId: req.user.\_id**(იგი მოდის მიდლვეარიდან, სადაც ვწვდებით მომხმარებლის დეითას):

1. const product = new Product({
2. title: title,
3. price: price,
4. description: description,
5. imageUrl: imageUrl,
6. userId: req.user
7. });

**13. Working with Mongoose**

220. One Important Thing About Fetching Relations

იმ შემთხვევაში, თუ გვსურს დეითას კონკრეტულ ფროფერთიზე უფრო დეტალური ინფო მივიღოთ, შეგვიძლია**.find()**-ს მივაბათ **.populate()** მეთოდი, რომელიც არგუმენტად იღებს სტრინგს, სადაც ეწერება ფროფერთის სახელწოდება:

1. exports.getProducts = (req, res, next) => {
2. Product.find()
3. .populate('userId')
4. .then((products) => {
5. res.render("admin/products", {
6. prods: products,
7. pageTitle: "Admin Products",
8. path: "/admin/products",
9. });
10. })
11. .catch(console.log);
12. };

სხვა შესაძლებლობებიც გვაქვს, თუნდაც **.pipulate()** მეთოდში შეგვიძლია მეორე არგუმენტად მივუთითოთ, რომელი დეითა გვჭირდება კონკრეტულად: **.populate('userId', 'name')**, ან გამოვიყენოთ მეთოდი**.select()**, რომელშიც ასევე კონკრეტული დეითას გამორიცხვაც შეგვიძლია - ნიშნის გამოყენებით, მაგალითად: **.select('title price -\_id')**.

**13. Working with Mongoose**

221. Working on the Shopping Cart

მონგუსის სქიმა ასევე საშუალებას გვაძლევს შევქმნათ ჩვენი მეთოდები, მაგალითად: **userSchema.methods.მეთოდისსახელი**. იგი აუცილებლად უნდა გაუტოლდეს **function () {}** ტიპის ფუნქციას, რათა იმუშავოს და გარკვეული ცვილილებების შემდეგ, შეგვიძლია ძველი კოდი ჩავაკოპიროთ ამ ფუნქციაში, რომელიც არგუმენტად მიიღებს შესაბამის დეითას:

*კოდი შემდეგში...*

**13. Working with Mongoose**

221. Working on the Shopping Cart

**user.js (model):**

1. userSchema.methods.addToCart = function (product) {
2. const cartProductIndex = this.cart.items.findIndex((cp) => {
3. return cp.productId.toString() === product.\_id.toString();
4. });
6. let newQuantity = 1;
7. const updatedCartItems = [...this.cart.items];
9. if (cartProductIndex >= 0) {
10. newQuantity = this.cart.items[cartProductIndex].quantity + 1;
11. updatedCartItems[cartProductIndex].quantity = newQuantity;
12. } else {
13. updatedCartItems.push({
14. productId: product.\_id,
15. quantity: newQuantity,
16. });
17. }
19. const updatedCart = {
20. items: updatedCartItems,
21. };
22. this.cart = updatedCart;
23. return this.save()
24. };

**13. Working with Mongoose**

222. Loading the Cart

ასე მივიღებდით კონკრეტულ ჩანესტებულ დეითას:

1. exports.getCart = (req, res, next) => {
2. req.user
3. .populate('cart.items.productId')
4. .then((user) => {
5. const products = user.cart.items
6. res.render("shop/cart", {
7. path: "/cart",
8. pageTitle: "Your Cart",
9. products: products,
10. });
11. })
12. .catch(console.log);
13. };

**13. Working with Mongoose**

227. Getting & Displaying the Orders

თუ კონკრეტული დეითას გამოძახებისას, ფროფერთის მივუთითებთ **\_doc**-ს, იგი სრულ ინფოს მოგვცემს დოკუმენტზე. მაგალითი:

1. exports.postOrder = (req, res, next) => {
2. req.user
3. .populate("cart.items.productId")
4. .then((user) => {
5. const products = user.cart.items.map((i) => {
6. return { quantity: i.quantity, product: { ...i.productId.\_doc } };
7. });
8. const order = new Order({
9. user: {
10. name: req.user.name,
11. userId: req.user,
12. },
13. products: products,
14. });
15. order.save();
16. })
17. .then(() => {
18. return req.user.clearCart();
19. })
20. .then(() => {
21. res.redirect("/orders");
22. })
23. .catch(console.log);
24. };

**14. Sessions & Cookies**

235. Setting a Cookie

**Cookie**- მისი გაგზავნე ბექიდან შეგვიძლია, რომელიც ბრაუზერის სთორში შეინახება. იგი იგზავნება შემდეგნაირად: res.setHeader('Set-Cookie', 'loggedIn=true') , სადაც **Set-Cookie** სტანდარტული დარეზერვებული სახელია, ხოლო მეორე არგუმენტად არის კი-ვალიუ წყვილები:

1. exports.postLogin = (req, res, next) => {
2. res.setHeader('Set-Cookie', 'loggedIn=true')
3. res.redirect('/');
4. }

მისი შენახვის მერე, ყველა რექვესტს თან ექნება ეგ **Cookie**, რომელსაც სერვერის მხარეს დავინახავთ.

აი, მაგალითად ასე:

1. exports.getlogin = (req, res, next) => {
2. const isLoggedIn = req.get("Cookie").split(";")[4].trim().split("=")[1];
4. res.render("auth/login", {
5. path: "/login",
6. pageTitle: "Login",
7. isAuthenticated: isLoggedIn,
8. });
9. };

**14. Sessions & Cookies**

236. Manipulating Cookies

**Cookie**-ებს ერთი მინუსი აქვთ ჩვენს შემთხვევაში: მისით მანიპულირება მომხმარებელს თავისუფლად შეუძლია, უბრალოდ სთორში შეცვლის მნიშვნელობას დდა ისე გამოგზავნის, მაგალითად მისი შემოწმებისას:

1. exports.getlogin = (req, res, next) => {
2. const isLoggedIn = req.get("Cookie").split(";")[4].trim().split("=")[1] === 'true';
4. res.render("auth/login", {
5. path: "/login",
6. pageTitle: "Login",
7. isAuthenticated: isLoggedIn,
8. });
9. };

**14. Sessions & Cookies**

237. Configuring Cookies

**Cookies**- ასევე იყენებენ მომხამრებლებზე თვალის სადევნად, რამდენადაც იგი მეხსიერებაში რჩება და შეუძლია მნიშვნელობის შეცვლა.

**Cookies**- ში ასევე სხვა არგუმენტებით შეგვიძლია მისი სიცოცხლის ხანგრძლივობის განსაზღვრა, მაგალითად **Expires**ან **Max-Age=10** (წამებში):

1. res.setHeader("Set-Cookie", "loggedIn=true; Max-Age=10");

**'Domain='** - ში ეითება ის დომეინი, სადაც ქუქი უნდა გაიგზავნოს (ასე უთვალთვალებენ)

**'Secure'** - მნიშვნელობის გაწერის გარეშე, გამოიყენება იმ მიზნით, რომ თუ გვერდი **https**-ს იყენებს, მხოლოდ მაშინ გამოიყენოს ეს ქუქი.

**'HttpOnly'** - კი Http-ისთვის გამაოიყენება.

**14. Sessions & Cookies**

239. Initializing the Session Middleware

რამდენადაც **Cookies**მომხმარებლის მხარეს ინახება, **Sessions**- უკვე სერვერის მხარეს მიდის და კონკრეტული სესიის ჰეშირებული აიდით ხდება გარკვევა, მომხმარებელი ეკუთვნის თუ არა კონკრეტულ სესიას.

*დანარჩენი ფოტოზე...*

სესიებით სარგებლობისთვის უნდა დავაინსტალიროთ პაკეჯი: npm install --save express-session

შემდეგ ვაიმპორტებთ და ვიყენებთ:

1. const session = require('express-session');
2. app.use(session({secret: 'my secret', resave: false, saveUninitialized: false, cookie: {maxAge: 10}}))

აქ **secret**-ში რასაც გვინდა ჩავწერთ, იგი ჰეშირებისთვის გამოიყენება.

**resave**-ის გათიშვა პერფორმანსის გაუმჯობესებაში დაგვეხმარება, რადგან ყოველჯერზე არ შექმნის ახალ სესიას, მხოლოდ მაშინ, თუ რამე შეიცვალა სესიაში.

**saveUninitialized**- იგი არაინიციალიზებული სესიის შენახვას გათიშავს.

**cookie**- იგი ქუქის ცონფიგურაციისთვისაა, თუმცა აუცილებელი არაა მისი გამოყენება.

**14. Sessions & Cookies**

240. Using the Session Middleware

ხოლო მას მერე, აღნიშნულ ფროფერთიზე **req**-ში ყველგან გვექნება წვდომა (**session**-ზე), რომლიდანაც თავისუფლად შევძლებთ ქუქიების დაყენებას და მერე მათზე მარტივად წვდომას. იგი ასევე ჰაშირებულ მნიშვნელობას შეინახავს ყოველ ახალ მომხამრებელზე:

დაყენება:

1. exports.postLogin = (req, res, next) => {
2. req.session.isLoggedIn = true;
3. res.redirect("/");
4. };

მიღება:

1. exports.getlogin = (req, res, next) => {
2. const isLoggedIn = req.session.isLoggedIn || false
3. res.render("auth/login", {
4. path: "/login",
5. pageTitle: "Login",
6. isAuthenticated: isLoggedIn,
7. });
8. };

**14. Sessions & Cookies**

241. Using MongoDB to Store Sessions

სესიების შენახვა სასურველია დეითაბეიზში. ჩვენ მას მონგოში შევინახავთ, თუმცა მანამდე პაკეჯი უნდა დავაინსტალიროთ:

npm install --save connect-mongodb-session

შემდეგ კი ძირითად ფაილში უნდა დავაიმპორტოთ და სესიის იმპორტის ობიექტთან დავაკავშიროთ შემდეგნაირად:

1. const session = require('express-session');
2. const MongoDBStore = require('connect-mongodb-session')(session);

შემდეგ კი რამდენადაც **MongoDBStore**კლასს წარმოადგენს, მის ახალ ინსტანსს შევქმნით და ჩავაწვდით ორ პარამეტრს, დეითაბეიზთან დაქონექთების ლინკს და კოლექციის სახელს:

1. const store = new MongoDBStore({
2. uri: MONGODB\_URI,
3. collection: 'sessions'
4. })

*\*შეიძლება ლინკიდან retryWrites=true წაშლა დაგვჭირდეს*

შემდეგ კი ჩვენს **app.use()**ში ჩავამატებთ store-ის პროფერთიში ამ ცვლადს:

1. app.use(
2. session({
3. secret: "my secret",
4. resave: false,
5. saveUninitialized: false,
6. store: store,
7. })
8. );

**14. Sessions & Cookies**

243. Deleting a Cookie

ამის მერე კი, სეშენის მიდლვეარში შეგვიძლია მომხმარებლის ან რაიმე სხვა ინფოს შენახვა, შემდეგ ყველგან რომ გამოვიყენოთ:

1. exports.postLogin = (req, res, next) => {
2. User.findById("619bd8f98fc91fd080263549")
3. .then((user) => {
4. req.session.isLoggedIn = true;
5. req.session.user = user;
6. res.redirect("/");
7. })
8. .catch(console.log);
9. };

**14. Sessions & Cookies**

243. Deleting a Cookie

თავად სესიის მიდლვეარში ჩაშენებულია მეთოდი **.destroy()**,რომელშიც ქოლბაქ ფუნქცია მიდის და გამოიძახება, როცა დასრულდება განადგურება. მისი გაშვების მერე სესია წაიშლება:

1. exports.postLogout = (req, res, next) => {
2. req.session.destroy((err) => {
3. console.log(err)
4. res.redirect('/')
5. })
6. };

სესიის ქუქი იქ დროებით მაინც დარჩება, თუმცა ტაბს რო დავხურავთ, ან ახალი შეიქმნება, იგი წაიშლება.

**14. Sessions & Cookies**

245. Making "Add to Cart" Work Again

რამდენადაც სესიის მიერ შენახული მომხმარებლის მონაცემები მხოლოდ მონაცემები იყო, ანუ ყველანაირი მონგუსის მეთოდი გარეშე, შემდეგნაირად გადავაკეთებთ და დავაკავშირებთ ორ **.use()**-ს:

1. app.use(
2. session({
3. secret: "my secret",
4. resave: false,
5. saveUninitialized: false,
6. store: store,
7. })
8. );
10. app.use((req, res, next) => {
11. if(!req.session.user) {
12. return next()
13. }
14. User.findById(req.session.user.\_id)
15. .then((user) => {
16. req.user = user;
17. next()
18. })
19. .catch(console.log);
20. })

**14. Sessions & Cookies**

246. Two Tiny Improvements

იმის გასაკონტროლებლად, რომ რედაირექტი სესიის შენახვაზე ადრე არ მოხდეს, შეგვიძლია სესიაზე **.save()**მეთოდი გამოვიყენოთ, დააა:

1. exports.postLogin = (req, res, next) => {
2. User.findById("619bd8f98fc91fd080263549")
3. .then((user) => {
4. req.session.isLoggedIn = true;
5. req.session.user = user;
6. req.session.save((err) => {
7. console.log(err)
8. res.redirect("/");
9. });
10. })
11. .catch(console.log);
12. };

**15. Adding Authentication**

255. Encrypting Passwords

პაროლის ჰეშირებისთვის გამოიყენება პაკეჯი - **bcryptjs**- რომელსაც ვაინსტალირებთ შემდეგნაირად: npm install --save bcryptjs

შემდეგ კი, საჭირო ფაილში დავაიმპორტებთ და გამოვიყენებთ მასზე მიბმულ მეთოდს -**.hash()**, რომლის პირველი არგუმენტია ის, რისი ჰაშირებაც გვსურს, და მეორე კი - რამდენჯერ მოხდეს ჰაშირების ციკლი:

1. const bcrypt = require('bcryptjs')
2. ...
3. bcrypt.hash(confirmPassword, 12)

**15. Adding Authentication**

257. Adding the Signin Functionality

ხოლო მოწოდებული პაროლის ჰაშირებულთან შედარება ხდება შემდეგი მეთოდით - **.compare()** :

1. User.findOne({ email: email })
2. .then((user) => {
3. if (!user) {
4. return res.redirect("/login");
5. }
6. bcrypt
7. .compare(password, user.password)
8. .then((doMatch) => {
9. if (doMatch) {
10. req.session.isLoggedIn = true;
11. req.session.user = user;
13. return req.session.save((err) => {
14. console.log(err);
15. res.redirect("/");
16. });
17. }
18. res.redirect("/login");
19. })
20. .catch(() => {
21. res.redirect("/login");
22. });
23. })
24. .catch(console.log);
25. };

**15. Adding Authentication**

259. Using Middleware to Protect Routes

**Middleware**- იგი სტანდარტულად იქმნება დაშემდეგ როუტის მომდევნო არგუმენტად გადაეცემა ნებისმიერ როუტს, რომლის დაცვაც გვსურს კონკრეტული ქონდიშენით, ან სხვა ინფოს გადაცემა აღმასრულებელი ფუნქციისთვის:

**middleware.js:**

1. module.exports = (req, res, next) => {
2. if(!req.session.isLoggedIn){
3. return res.redirect('/login')
4. }
5. next()
6. }

**მიმღები.js:**

1. const isAuth = require('../middleware/is-auth');
3. router.get("/add-product", isAuth, adminController.getAddProduct);
4. router.get("/products", isAuth, adminController.getProducts);

**15. Adding Authentication**

261. Using a CSRF Token

იმისათვის, რომ **CSRF**შეტევა არ მოხდეს (*ფოტოშია აღწერილი...*), უნდა დავაინსტალიროთ შემდეგი პაკეჯი: npm install --save csurf და დავაიმპორტოთ აპლიკაციის დონეზე, შემდეგ კი უბრალოდ ცვლადში შევინახოთ მასზე მიმაგრებული ფუნქცია - **csrf()**- შემდეგ კი აღნიშნული ცვლადი ჩავაწოდოთ **app.use()**-ს:

1. const csrf = require('csurf');
3. const csrfProtection = csrf();
5. app.use(csrfProtection);

ამის შემდეგ იგი ყველა **req**-ში ხელმისაწვდომი იქნება.

*გაგრძელდება...*

**15. Adding Authentication**

261. Using a CSRF Token

მას მაგალითად გამოვიყენებთ შოპის კონტროლერში შემდეგნაირად: **csrfToken: req.csrfToken()**, კოდში ასე:

1. exports.getIndex = (req, res, next) => {
2. Product.find()
3. .then((products) => {
4. res.render("shop/index", {
5. prods: products,
6. pageTitle: "Shop",
7. path: "/",
8. isAuthenticated: req.session.isLoggedIn,
9. csrfToken: req.csrfToken()
10. });
11. })
12. .catch(console.log);
13. };

აქედან გამომდინარე, იგი ხელმისაწვდომი იქნება **view**-ში, სადაც აუცილებლად უნდა ჩავაწოდოთ, ასევე სპეციალური სინტაქსით **value**ფროფერთიც, რომელსაც მნიშვნელობად ექნება **\_csrf**:

**navigation.ejs:**

1. <form action="/logout" method="post">
2. <input type="hidden" value="<%= csrfToken %>" name="\_csrf">
3. <button type="submit">Logout</button>
4. </form>

**15. Adding Authentication**

261. Using a CSRF Token

თუ ასე არ მოვიქცევით, **csrf**თოქენს ვერ ნახავს და ვალიდაცია არ დადასტურდება. იგი ავტომატურად, ყოველ ჯერზე, დააგენერირებს ახალ კოდს, რომელიც დაგვიცავს **CSRF**შეტევისგან

**15. Adding Authentication**

262. Adding CSRF Protection

თუ გვსურს რაიმე, მაგალითად**CSRF Protection**- მთელი აპლიკაციის მასშტაბით გამოვიყენოთ, მაგალითად როგორც **.render()** მეთოდის ობიექტის ნაწილი, შეგვიძლია აპლიკაციის დონეზე მიდლვეარში**.locals.**ფროფერთის წყალობით დავადეკლარიროთ მსგავსი რამ:

1. app.use((req, res, next) => {
2. res.locals.isAuthenticated = req.session.isLoggedIn;
3. res.locals.csrfToken = req.csrfToken()
4. next()
5. })

იგი როუტებამდე უნდა იყოს აუცილებლად (ძირითად როუტებამდე).

ამის მერე, ყველა როუტში, აღნიშნული ფროფერთები რენდერს ავტომატურად მიებმება, ანუ იქმნება ლოკალური ცვლადის მსგავსი რამ (ასე რომ, ხელით აღარ გავუწერთ).

შემდეგ კი, **View**-ებში საიდანაც **POST**რექვესტი იგზავნება, მსგავსი ინფუთი უნდა გვქონდეს, რათა ვალიდაცია მოხდეს:

1. <input type="hidden" value="<%= csrfToken %>" name="\_csrf">

**15. Adding Authentication**

264. Providing User Feedback

შეგვიძლია გამოვიყენოთ გარე პაკეტი იმისათვის, რომ მაგალითად რექვესტებს შორის მიმოვცვალოთ ერორის ინფორმაცია. ამისთვის ჯერ უნდა დავაინსტალიროთ ეს პაკეჯი: npm install --save connect-flash

შემდეგ აპლიკაციის დონეზე დავაიმპორტოთ და გამოყენებაში გავუშვათ:

1. const flash = require('connect-flash')
2. app.use(flash())

ხოლო შემდეგ კონკრეტულ როუტზე კონკრეტულ ექშენს მივამაგროთ:

1. if (!user) {
2. req.flash('error', 'Invalid email or password')
3. return res.redirect("/login");
4. }

ამის მერე კი იგი ხელმისაწვდომია ყველა რექვესტში. აღნიშნული ფუნქცია - **.flash()** - პირველ არგუმენტად იღებს კის, ხოლო მეორედ მესიჯს (სტრინგები). შემდეგ კი ამ კით მივწვდებით სხვა როუტში კონკრეტულ მესიჯს. მაგალითად:

1. res.render("auth/login", {
2. path: "/login",
3. pageTitle: "Login",
4. errorMessage: req.flash('error')
5. });

შემდეგ კი მას **view**-შიც გამოვიყენებთ.

**16. Sending Emails**

272. Using SendGrid

იმისათვის, რომ მეილის გაგზავნა შევძლოთ, უნდა ვისარგებლოთ მეილის სერვერით, რომლის ჩვენით შექმნაც ძალიან რთული იქნებოდა.

ამისთვის ჩვენ გამოვიყენებთ **SendGrid**-ის სერვერს.

მის გამოსაყენებლად, სასურველია დავაინსტალიროთ ორი პაკეჯი: **nodemailer**და **nodemailer-sendgrid-transport**:

npm install --save nodemailer nodemailer-sendgrid-transport

*დანარჩენი ფოტოზეა...*

**16. Sending Emails**

273. Using Nodemailer to Send an Email

შემდეგ კი საასურველ ლოკაციაზე დავაიმპორტებთ და გამოვიყენებთ.

1. const nodemailer = require("nodemailer");
2. const sendgridTransport = require("nodemailer-sendgrid-transport");

მათ კი ერთმანეთში ჩავაშენებთ:

1. const transporter = nodemailer.createTransport(
2. sendgridTransport({
3. auth: {
4. api\_key:
5. "...",
6. },
7. })
8. );

შემდეგ კი ამ ცვლადს გამოვიძახებთ:

1. transporter.sendMail({
2. to: email,
3. from: "gi.charkviani@gmail.com",
4. subject: "signup successed",
5. html: "<h1>Your successfully signed up!</h1>",
6. })

**17. Advanced Authentication**

278. Implementing the Token Logic

**Token**-ს ამ შემთხვევაში დავაგენერირებთ NodeJS-ის ბიბლიოთეკიდან - **crypto**:

1. const crypto = require('crypto')

ხოლო მას შემდეგნაირად გამოვიყენებთ:

1. crypto.randomBytes(32, (err, buffer) => {
2. if (err) return res.redirect("/");
3. const token = buffer.toString("hex");
4. })

**17. Advanced Authentication**

280. Creating the Reset Password Form

თუ გვსურს დეითაბეიზიდან მიღებული ინფორმაციის შედარება რამესთან, მაგალითად **Greater Than ($gt)** - შევიძლია მონგუსის სინტაქსით ასე დავწეროთ:

1. User.findOne({resetToken: token, resetTokenExpiration: {$gt: Date.now()}})

**17. Advanced Authentication**

286. Wrap Up

*აქ უბრალოდ აუთენთიფიკაცია გავაკეთეთ...*

**18. Understanding Validation**

291. Setup & Basic Validation

ვალიდაციისთვის გამოვიყენებთ **express-validator** პაკეჯს:

npm install --save express-validator

ობჯექტ დისტრაქშენით დავაიმპორტებთ როუტის ფაილში:

1. const { check } = require('express-validator/check');

შემდეგ კი, სასურველ მიდლვეარს შევარჩევთ და ჩავსვავთ, მაგალითად:

1. router.post('/signup', check('email').isEmail(), authController.postSignup);

ასევე თუ გვსურს, თავად კონტროლერებშიც გვქონდეს წვდომა იმ შემთხვევასთან, როცა დაერორდა და თავად ერორის მესიჯთან, მსგავსად ვიზამთ: შემოგვაქვს სხვა ფუნქცია - **validationResult**და მას არგუმენტად ვაწვდით უბრალოდ **req**-ს:

1. const { validationResult } = require("express-validator");
3. ...
4. if (!errors.isEmpty()) {
5. console.log(errors.array())
6. return res.status(422).render("auth/signup", {
7. path: "/signup",
8. pageTitle: "Signup",
9. errorMessage: errors.array(),
10. });
11. }
12. ...

**18. Understanding Validation**

292. Using Validation Error Messages

ასევე.**isEmail()**-ს შეგვიძლია დავუმატოთ მეთოდი **.withMessage(),** და აქ მივუთითოთ ის ერორ მესიჯი, რომელიც გამოჩნდება:

1. router.post('/signup', check('email').isEmail().withMessage('Please enter valid Email'), authController.postSignup);

**18. Understanding Validation**

293. Built-In & Custom Validators

ვალიდატორები, რომლებიც გამოვიყენეთ:

1. router.post('/signup', check('email').isEmail().withMessage('Please enter valid Email').custom((value, {req}) => {
2. if(value === 'test@test.com') {
3. throw new Error('This email address is forbidden!')
4. }
5. return true
6. }), authController.postSignup);

**18. Understanding Validation**

295. Checking For Field Equality

ვალიდაციის ახალი მეთოდები ახალ სტილში:

1. router.post(
2. "/signup",
3. [
4. check("email")
5. .isEmail()
6. .withMessage("Please enter valid Email")
7. .custom((value, { req }) => {
8. if (value === "test@test.com") {
9. throw new Error("This email address is forbidden!");
10. }
11. return true;
12. }),
13. body(
14. "password",
15. "Please enter a password with only numbers and text and at least 5 characters"
16. )
17. .isLength({ min: 5 })
18. .isAlphanumeric()
19. ],
21. authController.postSignup
22. );

**18. Understanding Validation**

295. Checking For Field Equality

წინას დამატება, პაროლების შედარება:

1. body('confirmPassword').custom((value, {req}) => {
2. if(value !== req.body.password) {
3. throw new Error('Passwords have to match!')
4. }
5. return true
6. })

**18. Understanding Validation**

296. Adding Async Validation

ასე გადავაკეთებთ ემეილის ბაზაში არსებობის კონფირმაციას:

1. check("email")
2. .isEmail()
3. .withMessage("Please enter valid Email")
4. .custom((value, { req }) => {
5. return User.findOne({ email: value })
6. .then((userDoc) => {
7. if (userDoc) {
8. return Promise.reject(
9. "Email exists already, please pick a different one"
10. );
11. }
12. });
13. }),

**18. Understanding Validation**

300. Sanitizing Data

სანიტარიზებისთვის შეგვიძლია გამოვიყენოთ მეთოდები: **.trim(), normalizeEmail()** და ა.შ.

**18. Understanding Validation**

302. Validating Product Editing

ასევე გამოვიყენეთ შემდეგი ტიპის ვალიდაციები:

1. router.post("/edit-product", [
3. body('title')
4. .isString()
5. .isLength({min: 3})
6. .trim(),
8. body('imageUrl')
9. .isURL(),
11. body('price')
12. .isFloat(),
14. body('description')
15. .isLength({min: 5, max: 400})
16. .trim()
18. ], isAuth, adminController.postEditProduct);

**19. Error Handling**

308. Errors - Some Theory

**Errors**:*მათზე ინფორმაცია ძირითადა ფოტოებშია...*

ცოტა რამ მათზე:

თუ გამოვიყენებთ **try{}catch(){}** ბლოკს, შესაძლებელი იქნება ერორის შემთხვევაშიც გავაგრძელოთ კოდის ეგზეკუშენი:

1. const sum = (a, b) => {
2. if (a && b) {
3. return a + b;
4. }
5. throw new Error("Invalid arguments");
6. };
8. try {
9. console.log(sum(1));
10. } catch (e) {
11. console.log('Error occured!');
12. }
14. console.log('This works!');

იგი გამოიყენება სინქრონული კოდისთვის, ხოლო ასინქრონულისთვის **.then().catch()** ბლოკი

**19. Error Handling**

311. Using the Express.js Error Handling Middleware

ერორ ჰენდლინგში გვეხმარება **Express**-ის ბიბლიოთეკა იმით, რომ იგი ცნობს ასევე ერორის მიდლვეარს, რომელიც იმ შემთხვევაში გამოიძახება, თუ **next()**-ს ჩავაწვდით არგუმენტს:

1. ...
2. .catch(err => {
3. return next(error)
4. });

ამის შემდეგ, შეგვიძლია app.js-ის დონეზე გავმართოთ შემდეგი მიდლვეარი:

1. app.use((error, req, res, next) => {
2. res.redirect('/500')
3. })

იგი, როგორც ვახსენე, გაეშვება ყველა ვარიანტში, თუ **next()**-ს აქვს არგუმენტი და თუ რამდენიმე მსგავსი გვაქვს, ზევიდან ქვევით გაეშვებიან (როგორც სტანდარტული მიდლვეარები).

**19. Error Handling**

313. Using the Error Handling Middleware Correctly

ასევე, კიდევ ერთი მნიშვნელოვანი ინფო ექსპრესის ერორის ჰენდლინგზე:

თუ ერორს რომელიმე **.use()**-ში **throw**-ს გავუკეთებთ სინქრონულად, ანუ არა ასინქრონულ ფუნქციაში, იგი ჩვეულებრივ გამოიძახებსდ შემდეგ ექსპრესის ერორ ჰენდლინგის ფუნქციას.

ხოლო თუ იგი ასინქრონულში ხდება, აუცილებლად**next()**-ს უნდა ჩავაწოდოთ **error**-ი:

1. app.use((req, res, next) => {
2. throw new Error('Sync Dummy') //გარედან, \*თუ ესაა, შიგა არ იმუშავებს
3. ...ასინქრონული ფუნქცია
4. .catch(err => {
5. next(new Error(err)) // შიგნიდან
6. });
7. })

ვინაიდან ლუპი არ გვსურს რომ შეიქმნას, როცა ერორი დაიჰენდლება, ეგრევე დარენდერდეს გვერდი (და არა რედაირექტი):

1. app.use((error, req, res, next) => {
2. res
3. .status(500)
4. .render("500", {
5. pageTitle: "Error",
6. path: "/500",
7. isAuthenticated: req.session.isLoggedIn,
8. });
9. })

**19. Error Handling**

314. Status Codes

**Server Status Codes URL:**

*https://httpstatuses.com/*

**20. File Upload & Download**

320. Handling Multipart Form Data

როდესაც ფაილის ატვირთვაზე მიდის საქმე, მნიშვნელოვანია ვიცოდეთ, რომ **.urlencoded()**, რომელ მიდლვეარსაც ვიყენებთ შემოსული ინფოს გასაშიფრად, ყველაფერს აქცევს ტექსტად (თავად **URLENCODED**არის ტექსტი). რამდენადაც ფაილი ბაინარია, იგი ტექსტად ვერ იქცევა და შესაბამისად, ვერც ბექში ვერ დავინახავთ მას:

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

 იმისათვის, რომ შემოსული რექვესტები დავაპარსიროდ ფაილად, ანუ ბაინარის გამოყენება შევძლოთ, უნდა დავაინსტალიროთ პაკეჯი **multer**:

npm install --save multer

ასევე მნიშვნელოვანია **HTML (EJX)**დავაკონფიგურიროთ. ანუ, **form**თაგში ჩავამატოთ ახალი ფროფერთ - **enctype**- რომელიც დეფაულტზე **application/x-www-form-urlencoded** არის, **multipart/form-data**-ად, რომელიც სერვერს შეატყობინებს, რომ ეს საბმიშენი, ეს რექვესტი, არა მხოლოდ უბრალო ტექსტს, არამედ ბაინარისაც შეიცავს.

ხოლო **multer**-ი რექვესტში ამ ტიპის დეითას ძიებას დაიწყებს და შემდეგ დააპარსირებს ჩვენს ინფოს.

**20. File Upload & Download**

321. Handling File Uploads with Multer

**Multer**გამოყენება:

იგი აპლიკაციის დონეზე უნდა დაიმპორტდეს და შემდეგ **app.use()**-ში გამოვიყენოთ, სასურველია **urlencoded()**-ის მერე

იგი გამოიძახება**.use()**-ში როგორც ფუნქცია, და მას ებმება სხვა მეთოდი, რომელიც დამოკიდებულია რა რაოდენობის ფაილს ვიღებთ, სტანდარტულად **.single()** - იგი კი არგუმენტად მიიღებს **name**-ად რაც გვაქვს **html**-ში გაწერილი, სტანდარტულად -**'image'**:

1. const multer = require('multer');
2. app.use(multer().single('image'))

შემდეგ კი მას სადაც გვურს, უბრალოდ **req.file**-ით მივწვდებით, რომელიც შეინახავს ინფორმაციას, ობიექტის სახით, რომელშიც იქნება თავად ფაილიც **buffer**ფროფერთიში, იგივე ფორმატით:

1. const imageUrl = req.file;

*გაგრძელდება...*

**20. File Upload & Download**

321. Handling File Uploads with Multer

ამ ობიექტის კონფიგურირებაც შეიძლება ძირითადი გამოძახების ლოკაციიდან, ანუ**app.js**, თავად **.multer()**-ში არგუმენტად ობიექტის ჩაწოდებით, წინასწარ განსაზღვრულ ფროფერთის გაწერით - **dest**- რომელიც მნიშვნელობად მიიღებს სტრინგს, რომელშიც **path**იქნება მითითებული, ანუ ფოლდერის სახელი აპლიკაციის დონეზე, სადაც უნდა შეინახოს ეს ფაილი. შედეგად კი, **multer**-ი ბაფერს ისევ ბაინარიში გადაიყვანს შეიანხავს მითითებულ ლოკაციაზე:

1. app.use(multer({dest: 'images'}).single('image'))

იგი შეინახება ბაინარად, ხოლო თუ ბოლოში გაფართოებას მივაწერთ თავად ფაილს, მაგ.**.png**, იგი ფოტოდ გადაიქცევა.

**20. File Upload & Download**

322. Configuring Multer to Adjust Filename & Filepath

უფრო მეტი კონფიგურირებისთვის, შეგვიძლია **multer()**-ში ჩაწოდებულ ობიექტს ერთი ფროფერთი გავუწეროთ - **storage**- ხოლო მისი მნიშვნელობა იყოს ობიექტი, რომელსაც ხელით დავაკონფიგურირებთ:

1. app.use(multer({ storage: fileStorage }).single("image"));

ცალკე შევქმნით **fileStorage**-ის სახელის მქონე ცვლადს, რომელიც გაუტოლდება **multer**-ს, მისი მეთოდითურთ - **diskStorage()**, რომელიც არგუმენტად იღებს ობიექტს თავისი პრედიფაინდ ფროფერთიებით, ესენია: **destination**და **filename**, რომლებიც მეთოდებს იღებენ მნიშვნელობებად, რომლებსაც აქვთ ერთნაირი ფროფერთიები - **req, file, cb** - სადაც **req**ტიპიური რექვესტია, **file**- თავად ფაილი, **cb**კი ქოლბაქ ფუნქცია, რომელიც პირველ არგუმენტად იღებს ერორის მნიშვნელობას, მეორედ კი თავად ინფოს, რაც უნდა დააბრუნოს.

*გაგრძელდება...*

**20. File Upload & Download**

322. Configuring Multer to Adjust Filename & Filepath

ასევ აღსანიშნავია, რომ თავად **file**-ი ის ობიექტია, რომელსაც **multer**აბრუნებს, როცა ფაილს იჭერს, ანუ მანდედან ნებისმიერ ფროფერთის შეგვიძლია მივწვდეთ.

**destination**- სად აიტვირთოს ფაილი,

**filename**- როგორ შეიცვალოს ფაილის სახელი

**fileStorage ცვლადი:**

1. const fileStorage = multer.diskStorage({
2. destination: (req, file, cb) => {
3. cb(null, "images");
4. },
5. filename: (req, file, cb) => {
6. cb(null, "myImages" + "-" + file.originalname);
7. },
8. });

**20. File Upload & Download**

323. Filtering Files by Mimetype

ფაილების გასაფილტრად ასევე ძირითად **multer**-ში კიდევ ერთი ფროფერთი შეგვიძლია გავხსნათ, რომელიც მარტივად - ფუნქცია იქნება, რომელსაც გარეთ დავაფორმირებთ. ეს ფროფერთია - **fileFilter**:

1. app.use(
2. multer({ storage: fileStorage, fileFilter: fileFilter }).single("image")
3. );

თავად **fileFilter**ცალკე ცვლადი იქნება, რომელშიც შეინახება ფუნქცია, სტანდარტული **multer**-ის ფუნქციის ფორმით, სადაც **file**ფროფეთის **mimeType**-ს მივწვდებით, სადაც ინახება ფაილის ტიპი, და მაგით შევამოწმებთ **cb**-ს მეორა არგუმენტად **true** ჰქონდეს, თუ **false**. ამის მიხედვით გადაწყდება ფაილი აიტვირთება თუ არა:

**fileFilter ცვლადი:**

1. const fileFilter = (req, file, cb) => {
2. if (
3. file.mimeType === "image/png" ||
4. file.mimeType === "image/jpg" ||
5. file.mimeType === "image/jpeg"
6. ) {
7. cb(null, true);
8. } else {
9. cb(null, false);
10. }
11. };

**20. File Upload & Download**

324. Storing File Data in the Database

ასევე, მას შემდეგ, რაც დამატებითი ობიექტი ჩავაწოდეთ **multer**-ის ძირითად გამოყენების ფუნქციას, **Buffer**-ის ჩანაცვლა ფიზიკურმა ლოკაციამ ფაილის, ანუ **path**გვაქვს ფროფერთი **multer**-ის ობიექტში...

ხოლო ამ ობიექტში **path**-ს შემდეგნაირად მივწვდებოდით ნებისმიერ კონტროლერში:

1. const image = req.file;
2. const imageUrl = image.path;

**20. File Upload & Download**

326. Serving Images Statically

**static serving** ფოლდერისთვის ნიშნავს, რომ ფრონტის რექვესტები ავტომატურად დაიჰენდლება **express**-ის მიერ, ანუ თუ ფრონტი რექვესტს აკეთებს ფაილზე, რომელიც ჩვენს სტატიკურ  **public**ფოლდერშია, იგი ავტომატურად გაეგზავნება.

იგივენაირად უნდა მოვექცეთ ფოტოების ფოლდერსაც

ანუ, როდეასც ჩვენი **HTML**წაიკითხება, სადაც ფოტოების src-ში ლინკია მითითებული, სადაც რექვესტი უნდა გაკეთდეს*(სტანდარტულად ჯობს წინ****/****ჰქონდეთ, რათა აბსოლუტური, ანუ არსებული****url****-ი გმაოიყენოს ყოველგვარი უკვე არსებული პარამეტრების გარეშე და შემდეგ მას მიაბას ფოტოს ლინკი, რომელიც დაემთხვევა ჩვენს სტატიკურ ფოლდერს და ფოტოს გავუგზავნით)*:

**HTML:**

<img src="/<%= product.imageUrl %>" alt="<%= product.title %>">

*გაგრძელდება...*

**20. File Upload & Download**

326. Serving Images Statically

შემდეგ კი, აპლიკაციის დონეზე, სადაც სტატიკურად ვემსახურებით, ასევე ჯობს მივუთითოთ, რომ თუ**"/images"** ლინკზე შემოვა რექვესტი, გამოიყენოს კონკრეტული სტატიკური ფოლდერი, რომელიც თავის თავში გულისხმობს იმას,რომ რომელ ფოლდერსაც ვემსახურებით, იგი განიხილება **root**დონეზე:

1. app.use("/images", express.static(path.join(\_\_dirname, "images")));

**20. File Upload & Download**

328. Setting File Type Headers

ჰედერების, ფაილური სისტემისა და path-ის წყალობით, შეგვიძლია ფაილი გავუგზავნოთ უკან მომხმარებელს, მაგალითად ინვოისი

მაგ.:

1. exports.getInvoice = (req, res, next) => {
2. const orderId = req.params.orderId;
3. const invoiceName = "invoice-" + orderId + ".pdf";
4. const invoicePath = path.join("data", "invoices", invoiceName);
5. fs.readFile(invoicePath, (err, data) => {
6. if (err) {
7. return next(err);
8. }
9. res.setHeader("Content-Type", "application/pdf")
10. res.setHeader('Content-Disposition', 'attachment; filename="' + invoiceName + '"')
12. res.send(data);
13. });
14. };

აქედან:

ჰედერები განსაზღვრავენ ბრაუზერის ქმედებას, ანუ როგორ მოექცეს კონკრეტულ ფაილს. attachment - ეუბნება, რომ ჩამოტვირთოს ფაილი პირდაპირ, ხოლო თუ მის ნაცვლად '**inline'**-ს გავწერდით, იქვე გახსნიდა ფაილს, ბრაიზერშივე. ასევე ამ უკანასკნელს აკეთებსapplication/pdf, თუ ბრაუზერს შეუძლია ამის დაჰენდლვა, ასევე იგივეა Content-Disposition - კი კონკრეტულად ჰენდლინგის განმსაზღვრელი ჰედერია.

**20. File Upload & Download**

330. Streaming Data vs Preloading Data

რამდენადაც **fs.readFile()** არ იქნებოდა კარგი მიდგომა სერვერის მხარეს, ვინაიდან ბევრი რექვესტის შემთხვევაში და იმ შემთხვევაშიც, თუ დიდი ფაილია, იგი სერვერის მეხსიერების გადატვირთვას გამოიწვევდა.

ამის ალტერნატივაა**fs.createReadStream()**, რომელიც ასევე არგუმენტად **path**-ს იღებს და წინასგან განსხვავებით, იგი ნაწილ-ნაწილ წაიკითხავს ფაილს.

რამდენადაც **res**ობიექტი წარმოადგენს ჩაწერად, იგივე **writeable**ობიექტს, შეგვიძლია ამ კონკრეტული ფუნქციაში არსებული**.pipe()** მეთოდი გამოვიყენოთ და მას ჩავაწოდოთ **res**არგუმენტად. შედეგად კი იგი, ნაწილ-ნაწილ, ჩანკებად, გაუგზავნის ფაილს მომხმარებელს ისე, რომ სხვა რექვესტები არ დაყოვნდება და არც მეხსიერება გადაიტვირთება.

შეცვლილი სიმეთოდით:

1. const file = fs.createReadStream(invoicePath);
2. res.setHeader("Content-Type", "application/pdf");
3. res.setHeader(
4. "Content-Disposition",
5. 'inline; filename="' + invoiceName + '"'
6. );
8. file.pipe(res)

**20. File Upload & Download**

331. Using PDFKit for .pdf Generation

**PDFKit**- პაკეჯი გამოიყენება **PDF**-ების შესაქმნელად. მისი დოკუმენტაცია იყენებს ე.წ. **CoffeScript**, რომელიც**JS**-ის სუპერსეტია.

მას შემდეგნაირად ვაინსტალირებთ: npm install --save pdfkit

ხოლო შემდეგ, სადაც მისი გამოყენება გვსურს, დავაიმპორტებთ და შემდეგ მის ახალ ინსტანსს შევინახავთ ცვლადში:

1. const PDFDocument = require('pdfkit')
2. const pdfDoc = new PDFDocument();

იგი ასევე წარმოადგენს წაკითხვათ სტრიმს, შედეგად მას აქვს მეთოდი**.pipe()**, სადაც შეგვიძლია ჩავაწოდოთ ჩვენი **fs.createWriteStream()** ფუნქცია არგუმენტად.

და ასევე ჩავაწდოდოთ, როგორც წინაზე, **res**, როგორც არგუმენტი და არა მხოლოდ ფაილურ სისტემაში შეინახება შექმნილი **PDF**ფაილი, არამედ მომხმარებელსაც გაეგზავნება.

*გაგრძელდება...*

**20. File Upload & Download**

331. Using PDFKit for .pdf Generation

შემდეგ, მის შესაქმნელად, ერთ-ერთი მეთოდია **.text()**, რომელიც სტრინგს მიიღებს არგუმენტად, რომელიც **pdf**-ში ჩაიწერება.

ხოლო როცა დავასრულებთ მის შექმნას, უნდა გამოვიძახოთ ფუნქცია**.end()**, რომ მიხვდეს და საბოლოოდ შექმნას და შეინახოს ფაილი

მაგ.:

1. const pdfDoc = new PDFDocument();
2. res.setHeader("Content-Type", "application/pdf");
3. res.setHeader(
4. "Content-Disposition",
5. 'inline; filename="' + invoiceName + '"'
6. );
8. pdfDoc.pipe(fs.createWriteStream(invoicePath))
9. pdfDoc.pipe(res);
11. pdfDoc.text('Hello world!')
12. pdfDoc.end()

**20. File Upload & Download**

332. Generating .pdf Files with Order Data

ასეთი **pdf**შევქმენით:

1. const pdfDoc = new PDFDocument();
2. res.setHeader("Content-Type", "application/pdf");
3. res.setHeader(
4. "Content-Disposition",
5. 'inline; filename="' + invoiceName + '"'
6. );
8. pdfDoc.pipe(fs.createWriteStream(invoicePath));
9. pdfDoc.pipe(res);
11. pdfDoc.fontSize(26).text("Invoice", {
12. underline: true,
13. });
15. pdfDoc.text("------------------------");
16. let totalPrice = 0;
17. order.products.forEach((prod) => {
18. totalPrice += prod.quantity \* prod.product.price;
19. pdfDoc
20. .fontSize(14)
21. .text(
22. prod.product.title +
23. " - " +
24. prod.quantity +
25. " x " +
26. "$" +
27. prod.product.price
28. );
29. });
30. pdfDoc.text('---')
31. pdfDoc.fontSize(20).text("Total Price: $" + totalPrice);
33. pdfDoc.end();

**20. File Upload & Download**

333. Deleting Files

**fs**-ით ფაილის წაშლა ხდება მეთოდით - **unlink()**, რომელიც პირველ არგუმენტად იღებს მისამართს ფაილის, ხოლო მეორედ ქოლბაქს ერორისთვის.

იმპლემენტაცია:

1. const fs = require('fs');
3. exports.deleteFile = (filePath) => {
4. fs.unlink(filePath, (err) => {
5. if(err) throw err;
6. })
7. }

**21. Adding Pagination**

339. Retrieving a Chunk of Data

**QueryParams**ნოუდში შემდეგნაირად ვწვდებით:

1. exports.getIndex = (req, res, next) => {
2. const page = req.query.page;
3. }

**mongoose**-ს აქვს **pagination**-ისთვის საჭირო 2 მეთოდი: **.skip()** და**.limit()**, რომლებიც არგუმენტებად რიცხვებს იღებენ.

**.skip()**- რა რაოდენობის დეითა უნდა გამოტოვოს

**.limit()** - რა რაოდენობის დეითა წამოიღოს მაქსიმუმ

მაგალითად:

1. exports.getIndex = (req, res, next) => {
2. const page = req.query.page;
4. Product.find()
5. .skip((page - 1)\*ITEMS\_PER\_PAGE)
6. .limit(ITEMS\_PER\_PAGE)
7. .then((products) => {
8. res.render("shop/index", {
9. prods: products,
10. pageTitle: "Shop",
11. path: "/",
12. });
13. })
14. ...
15. };

**21. Adding Pagination**

341. Preparing Pagination Data on the Server

**SQL:**

Here's how you would implement pagination in SQL code: https://stackoverflow.com/questions/3799193/mysql-data-best-way-to-implement-paging

To quickly sum it up: The LIMIT command allows you to restrict the amount of data points you fetch, it's your limit() equivalent. Combined with the OFFSET command (which replaces skip()), you can control how many items you want to fetch and how many you want to skip.

When using Sequelize, the official docs describe how to add pagination: https://sequelize.org/master/manual/model-querying-basics.html

**21. Adding Pagination**

341. Preparing Pagination Data on the Server

იმისათვის, რომ დოკუმენტების რაოდენობა მივიღოთ **MongoDB**-დან, **mongoose**გვთავაზობს შემდეგ მეთოდს: **countDocuments()**:

1. Product.find()
2. .countDocuments()
3. .then((numProducts) => {
4. totalItems = numProducts;
5. return Product.find()
6. .skip((page - 1) \* ITEMS\_PER\_PAGE)
7. .limit(ITEMS\_PER\_PAGE);
8. })

**22. Understanding Async Requests**

347. What are Async Requests?

**Data (JSON)** - ის მიმოცვლით აღარაა საჭირო ყოველ ჯერზე ახალი **HTML**-ის გაგზავნა ან დარენდერება, მაგის ნაცვლად ვაგზავნით **JSON**დეითას და დინამიურად ვანახლებთ არსებულ **HTML**-ს.

*დანარჩენი ფოტოზეა...*

**22. Understanding Async Requests**

350. Sending & Handling Background Requests

რამდენადაც **csrf**ბიბლიოთეკა არა მხოლოდ **body**-ში, არაედ **headers**-შიც იყურება, რათა კოდი შეამოწმოს, ფრონტიდან რექვესტის გაგზავნისას **DELETE**რექვესტ მაგალითად შეგვიძლია ჰედერად მივაბათ **csrf**კოდი (რადგან მას body არ აქვს).

**csrf**ჰედერებში მის ძებნას დაიწყებს -**'csrf-token'**- ჰედერში:

1. const deleteProduct = (btn) => {
2. const prodId = btn.parentNode.querySelector('[name=productId]').value
3. const csrf = btn.parentNode.querySelector('[name=\_csrf]').value
5. fetch('/admin/product/' + prodId, {
6. method: 'DELETE',
7. headers: {
8. 'csrf-token': csrf
9. }
10. })
11. }

**22. Understanding Async Requests**

351. Manipulating the DOM

**Client-side JS for deleting:**

1. const deleteProduct = (btn) => {
2. const prodId = btn.parentNode.querySelector('[name=productId]').value
3. const csrf = btn.parentNode.querySelector('[name=\_csrf]').value
4. const productElement = btn.closest('article')
6. fetch('/admin/product/' + prodId, {
7. method: 'DELETE',
8. headers: {
9. 'csrf-token': csrf
10. }
11. })
12. .then(result => {
13. return result.json()
14. })
15. .then(data => {
16. console.log(data)
17. productElement.remove()
18. })
19. .catch(console.log)
20. }

**23. Adding Payments**

356. Using Stripe in Your App

**Stripe**დოკუმენტაცია:*https://stripe.com/docs/payments*

**JS**-ისთვის: *https://stripe.com/docs/js*

**Server-side:**npm install --save stripe -*ინსტალაციისთვის მხოლოდ...*

**Client-side:**

1. <script src="https://js.stripe.com/v3/"></script>
2. <script>
3. var stripe = Stripe('pk\_test\_...');
4. var orderBtn = document.getElementById('order-btn');
5. orderBtn.addEventListener('click', function() {
6. stripe.redirectToCheckout({
7. sessionId: '' // ეს გაგრძელდება
8. })
9. })
10. </script>

*გაგრძელდება...*

**23. Adding Payments**

356. Using Stripe in Your App

**Server:**

**Stripe**-ის ინსტალაციის შემდეგ, შეგვიძლია კონკრეტულ კონტროლერში დავაიმპორტოთ სტრაიპის ბიბლიოთეკა, არგუმენტად ეგრევე ჩავაწოდოთ ე.წ. **Secret key**, რომელიც ჩვენს გვერდზეა.

შემდეგ ის ცვლადი, რომელშიც ამ იმპორტს ვინახავთ გამოგვადგება იმისთვის, რომ **session id** დავაგენერიროთ.

ფუნქცია ასე უნდა დაკონფიგურირდეს:

1. const stripe = require('stripe')('sk\_test\_...')
2. ...
3. return stripe.checkout.sessions.create({
4. payment\_method\_types: ['card'],
5. line\_items: products.map(p => {
6. return {
7. name: p.productId.title,
8. description: p.productId.description,
9. amount: p.productId.price \* 100,
10. currency: 'usd',
11. quantity: p.quantity
12. }
13. }),
14. success\_url: req.protocol + '://' + req.get('host') + '/checkout/success',
15. cancel\_url: req.protocol + '://' + req.get('host') + '/checkout/cencel'
16. })
17. ...

**success\_url**და **cencel\_url**-ში გაწერილი კონფიგურირების მეთოდები, ლოგიკურად, **req**-ში უკვე არსებულია და შეიძლება შესაბამისად მათი გამოყენება ნებისმიერ დროს.

**23. Adding Payments**

356. Using Stripe in Your App

ასევე **მნიშვნელოვანია**: სახელი ერქვას ჩვენს ექაუნთს, რომ არ დერორდეს.

**Client-side** ისევ, უკვე დააფდეითებული **session id**-ით, რომელსაც ბექიდან ვიღებთ და შემდეგრენდერის ობიექტში ვუშვებთ:

1. <script src="https://js.stripe.com/v3/"></script>
2. <script>
3. var stripe = Stripe('pk\_test\_...');
4. var orderBtn = document.getElementById('order-btn');
5. orderBtn.addEventListener('click', function() {
6. stripe.redirectToCheckout({
7. sessionId: '<%= sessionId %>'
8. })
9. })
10. </script>

**24. Working with REST APIs - The Basics**

359. What are REST APIs and why do we use Them?

ძირითად იგანსხვავება კლასიკურ მიდგომასა (**rendering**სერვერზე) **REST API**-ს შორის (**Representational State Transfer**) არის სერვერის მიერ დაბრუნებულ პასუხში. უკანასკნელის შემთხვევაში სერვერი არა **HTML**-ს, არამედ უბრალოდ დეითას უგზავნის, ძირითადად **JSON**ფორმატით და შემდეგ **Client-side** JS არენდერებს ახალ **HTML**-ს დეითას მიხედვით.

*დანარჩენი ფოტოებზეა...*

**24. Working with REST APIs - The Basics**

361. Understanding Routing & HTTP Methods

აქამდე, ტრადიციული **web-app**-ის მიდგომით, სადაც სერვერზე რენდერდება **HTML**და ისე იგზავნება, ფრონტიდან კომუნიკაცია ხდება **forms**თაგის საშუალებით თუნდაც, ანდაც უბრალოდ როუტინგით, სადაც a თაგს ვიყენებთ.

**\*form** - ამისთვის ვიყენებდით ბექში **urlencoded**-საც

**JS**-იდან ძირითადად, როცა რექვესტს ვაგზავნით, იგი იღებს კონკრეტულ დეითას და მისი მიხედვით შემდეგ არენდერებს ახალ **HTML**-ს.

*დანარჩენი ფოტოებზე...*

**24. Working with REST APIs - The Basics**

363. Creating our REST API Project & Implementing the Route Setup

ახალი პროექტი დავიწყეთ **REST API**-სთვის.

მნიშვნელოვანია ვიცოდეთ, რომ ცალკე ბიბლიოთეკებიც არსებობს **express**-ის გარდა, რომლებიც კონკრეტულად**REST API**-ს ემსახურებიან, თუმცა აქ **express**გამოვიყენებთ.

*დავაკონსპექტებ მხოლოდ იმ თემებს, რომლებიც ახალია...*

**24. Working with REST APIs - The Basics**

364. Sending Requests & Responses and Working with Postman

სტანდარტულად, გამოვიყენებთ შემდეგი ტიპის რესპონსს, **.json()**, რომელიც არგუმენტად სტანდარტულად მიიღებს ობიექტს და ავტომატურად დააფორმატირებს **JSON**-ში და ისე გაგზავნის:

1. exports.getPosts = (req, res, next) => {
2. res.status(200).json({
3. posts: [{title: 'First Post', content: 'This is the first post!'}]
4. })
5. };

ასევე შემოსული დეითას (**POST**) პარსირებას მოვახდენთ **express.json()**-ით, რომელიც **url.encoded()**-ისგან განსხვავებით, რომელიც იყენებდა ფორმატს - **x-www-form-urlencoded***(multer-ისთვის სხვა ფორმატს)*-გამოვიყენებთ:**application/json**-ს.

1. app.use(express.json())

დეითას ისევ**req.body**-ში მივწვდებით, როგორც აქამდე:

1. exports.createPost = (req, res, next) => {
2. const title = req.body.title;
3. const content = req.body.content;
4. ...
5. }

**24. Working with REST APIs - The Basics**

364. Sending Requests & Responses and Working with Postman

ბრაუზერით შეგვიძლია **GET**რექვესტი გავაკეთოთ **REST API**-ზე და წავიკითხოთ მიღებული დეითა (**JSON**ძირითადად), თუმცა მისი უკეთესი ალტერნატივაა ე.წ.  **Postman**, რომელიც არა მხოლოდ **GET**რექვესტებს, არამედ ყველა ტიპის რექვესტის დაჰენდლვა შეუძლია.

ლინკი: *https://www.postman.com/*

მანდედან **GET**-ის გაგზავნა მარტივად ხდება.

**POST**-ისას: ***body>raw>JSON***

**24. Working with REST APIs - The Basics**

365. REST APIs, Clients & CORS Errors

**CORS - Cross-Origin Resource Sharing**- იგი ერორს აგდებს ხშირად, როცა **Single Page app**-ზე მუშაობ, რადგან იგი დეფაულტზე ბრაუზერების მიერ არაა დაშვებული.

ანუ, ბრაუზერში ჩაშენებულია უსაფრთხოების სისტემა, რომელიც არ გაძლევს საშუალებას, გარდა იმ სერვერისა, რომელზეც შენი აპია, გაგზავნო რექვესტი, ანუ გააზიარო ინფორმაცია.

თუმცა, ამაზე გადაწერა შესაძლებელია. ანუ ბრაუზერს უნდა ვუთხრათ, რომ მიიღოს ჩვენი სერვერის მიერ გამოგზავნილი რესპონსი.

მისი მოგვარება შესაძლებელია მხოლოდ სერვერის მხარეს.

*გაგრძელდება...*

**24. Working with REST APIs - The Basics**

365. REST APIs, Clients & CORS Errors

ამისთვის საჭიროა **Headers**-ის დაყენება. 3 ტიპის ჰედერია საჭირო: ზოგადი დაშვების, **HTTP**მეთოდების დაშვებისა და თავად ჰედერების ტიპების დაშვების. *იგი ყველაზე მაღლა უნდა იდგეს app.use()-ებში*. მაგალითი:

1. app.use((req, res, next) => {
2. res.setHeader('Access-Control-Allow-Origin', '\*') // იგი შემოუშვებს კონკრეტულ ან ნებისმიერ (\*) უცხო რექვესს
3. res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, PATCH, DELETE') // იგი ნებას დართავს უცხო რექვესტებს, რომ გამოიყენონ კონკრეტული მეთოდები
4. res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization') // იგი დააწესებს ან მოთხოვს, სპეციალურ ჰედერებს რომ შეიცავდნენ რექვესტები, რათა პასუხი დაუბრუნოს. აქაც შესაძლოა \*-ის გამოყენება
5. next()
6. })

**24. Working with REST APIs - The Basics**

365. REST APIs, Clients & CORS Errors

აღნიშნული პრობლემა აქამდე არ წარმოიშობოდა, რადგან რომელ სერვერზეც იყო საიტი, იმ სერვერზევე ხდებოდა რენდერინგი, ანუ რექვესტების მიღება.

**24. Working with REST APIs - The Basics**

366. Sending POST Requests

**Request Method: OPTIONS** - ძირითადად იგზავნება ბრაუზერის მიერ, რათა შეამოწმოს, კონკრეტული მეთოდი დაშვებულია თუ არა სერვერზე. იგი შეიძლება შეგვხვდეს მაგალთად **POST**რექვესტის გაგზავნისას, სადაც 2 რექვესტი იქნება გაგზავნილი, ერთი **POST**და მეორე **OPTIONS**.

შემდეგნაირად გავგზავნიდით ფრონტიდან რექვესტს:

1. postButton.addEventListener('click', () => {
2. fetch('http://localhost:8080/feed/post', {
3. method: 'POST',
4. body: JSON.stringify({
5. title: 'A Codepen Post',
6. content: 'Created via Codepan'
7. }),
8. headers: {
9. 'Content-Type': 'application/json'
10. }
11. })
12. .then(result => {
13. return result.json()
14. })
15. .then(jsonData => {
16. console.log(jsonData)
17. })
18. .catch(console.log)
19. })

**25. Working with REST APIs - The Practical Application**

376. Setting Up a Post Model

**mongoose**-ის **Schema**-ს კონსტრუქტორს შეგვიძლია მეორე არგუმენტად ჩავაწოდოთ ასევე **timestamps: true** ობიექტი, რომელიც ყველა აითემს თაიმსტამპს მიაბამს:

1. const mongoose = require('mongoose');
3. const Schema = mongoose.Schema;
5. const postSchema = new Schema({
6. title: {
7. type: String,
8. required: true
9. },
10. imageUrl: {
11. type: String,
12. required: true
13. },
14. content: {
15. type: String,
16. required: true
17. },
18. creator: {
19. type: Object,
20. required: true
21. }
22. }, { timestamps: true})

**25. Working with REST APIs - The Practical Application**

381. Uploading Images

Image Names & Windows

In the next lecture, you'll learn how to upload images in a REST world.

One **important** note for**Windows users only**:

On Windows, the file name that includes a date string is not really supported and will lead to some strange CORS errors. Adjust your code like this to avoid such errors:

Instead of

1. const storage = multer.diskStorage({
2. destination: function(req, file, cb) {
3. cb(null, 'images');
4. },
5. filename: function(req, file, cb) {
6. cb(null, new Date().toISOString() + file.originalname);
7. }
8. });

which we'll write in the next lecture, you should use this slightly modified version:

1. const { v4: uuidv4 } = require('uuid');
3. const storage = multer.diskStorage({
4. destination: function(req, file, cb) {
5. cb(null, 'images');
6. },
7. filename: function(req, file, cb) {
8. cb(null, uuidv4())
9. }
10. });

*გაგრძელდება...*

**25. Working with REST APIs - The Practical Application**

381. Uploading Images

For this, install the uuid package by running:

npm install --save uuid

To ensure that images can be loaded correctly on the frontend, you should also change the logic in the feed.js controller:

in createPosts, change the imageUrl const:

1. exports.createPost = (req, res, next) => {
2. ...
3. const imageUrl = req.file.path.replace("\\" ,"/");
4. ...
5. }

and in updatePost (once we added that later):

1. exports.updatePost = (req, res, next) => {
2. ...
3. imageUrl = req.file.path.replace("\\","/");
4. }

On macOS and Linux, you can ignore that and stick to the code I show in the videos.

**25. Working with REST APIs - The Practical Application**

381. Uploading Images

**Front-End File Upload:**

**JS-**ს აქვს ჩაშენებული ობიექტი კონსტრუქტორი, რომელიც საშუალებას გვაძლევს **Form Data** დავაგენერიროთ ( **new FormData()**).

მასში არსებობს მეთოდი**.append()**, რომელიც ამ საქმეს აკეთებს. იგი პირველ არგუმენტად იღებს სახელს, მეორედ კი თავად შემცველობას, ანუ მნიშვნელობას (იგივეა რაც, ინფუთისას - **name**და **value**):

1. const formData = new FormData();
2. formData.append('title', postData.title)
3. formData.append('content', postData.content)
4. formData.append('image', postData.image)

ასევე ფრონტის მხარეს ჰედერის დაყენება აღარაა საჭირო, რადგან **FormData**ავტომატურად დააყენებს მას.

რექვესტი:

1. fetch(url, {
2. method: method,
3. body: formData
4. })

**25. Working with REST APIs - The Practical Application**

390. Logging In & Creating JSON Web Tokens (JWTs)

**Json Web Token - JWT -** პაკეჯი გვეხმარება თოქენების შექმნაში.

მის დასაინსტალირებლად: npm install --save jsonwebtoken

მას ჯერ ვაიმპორტებთ და შემდეგ ვინახავთ ცვლადში მის ფუნქციიას, რომელსაც არგუმენტებს ვაწვდით და საბოლოო ჯამში აბრუნებს **token**-ს.

პირველი არგუმენტია კონფიგურაციისთვის, მეორე საიდუმლო სტრინგი, მესამე კი ვადის გასვლის ობიექტი:

1. const jwt = require('jsonwebtoken');
2. const token = jwt.sign(
3. {
4. email: loadedUser.email,
5. userId: loadedUser.\_id.toString(),
6. },
7. "secret",
8. {expiresIn: "1h"}
9. );

*დანარჩენი ფოტოზე...*

**25. Working with REST APIs - The Practical Application**

390. Logging In & Creating JSON Web Tokens (JWTs)

*https://jwt.io/*- აქ შეიძლება **token**-ის გაჩიფვრა

**25. Working with REST APIs - The Practical Application**

391. Using & Validating the Token

მას შემდეგ, რაც **Token**-ს დავუგენერირებთ და გავუგზავნით მომხმარებელს (ავტორიზაციის შემდეგ), მომხმარებელმა ვალიდაციისთვის მის ყველა რექვესტს უნდა მიაბას ეს **Token**-ი, რომელსაც იგი შეინახავს.

მისი მიბმა შეეძლო, როგორც **queryParams**-ში, ასევე body-ში (თუმცა **GET**-ს არ აქვს ეგ).

იდეალური ვარიანტია ჰედერში ჩასმა **Token**-ის. ჰედერისთვის ნებისმიერი რამ შემეძლო დამერქვა, თუმცა სასურველია ოფიციალური სათაური - **Authorization** - გამოვიყენო (ბექშიც უნდა დაუშვა ეგ ჰედერი **setHeader()**-ში)

კონვენციის თანახმად, სასურველია თოქენი დაიწყოს **'Bearer '**-ით:

1. headers: {
2. Authorization: 'Bearer ' + this.props.token
3. }

შემდეგ კი ბექის მხარეს იგივე პაკეჯში არსებულ **.verify()** მეთოდს გამოვიყენებ, რომელიც პირველ არგუმენტად იღებს token-ს, ხოლო მეორე არგუმენტად - საიდუმლო სტრინგს. (ასევე შემეძლო .decode() გამომეყენებინა, თუმცა იგი შიგნით არსებულ ინფოს არ მომაწვდის):

1. jwt.verify(token, "secret")

*გაგრძელდება...*

**25. Working with REST APIs - The Practical Application**

391. Using & Validating the Token

**JWT**-ს მიდლვეარი იქნებოდა კარგი გამოსავალი, რათა ის მიბმოდა რექვესტებს, სადაც აუთენთიფიკაციაა საჭირო:

**isAuth.js (middleware):**

1. const jwt = require("jsonwebtoken");
3. module.exports = (req, res, next) => {
4. const token = req.get("Authorization").split(" ")[1];
5. let decodedToken;
6. try {
7. decodedToken = jwt.verify(token, "secret");
8. } catch (err) {
9. err.statusCode = 500;
10. throw err;
11. }
12. if(!decodedToken) {
13. const error = new Error('Not authenticated.')
14. error.statusCode = 401;
15. throw error
16. }
17. req.userId = decodedToken.userId;
18. next()
19. };

შემდეგ კი, სადაც საჭიროა, მაგალითად:

1. const isAuth = require('../middleware/isAuth');
3. router.get("/posts", isAuth, feedController.getPosts);

**25. Working with REST APIs - The Practical Application**

395. Clearing Post-User Relations

**Mongoose**გვთავაზობს მეთოდს **.pull()**, რომელიც მასივს შეგვიძლაი მივაბათ და რგუმენტად ვაწვდით კონკრეტული აითემის აიდის და იგი მას პოულობს და შლის:

1. ...
2. .then(user => {
3. user.posts.pull(postId)
4. return user.save()
5. })
6. ...

**26. Understanding Async Await in Node.js**

400. Transforming "Then Catch" to "Async Await"

**Async..Await** - საბოლოო ჯამში ისევ**.then()** ჯაჭვამდე დადის, თუმცა მას ჩვენ, რა თქმა უნდა, ვერ ვხედავთ.

იგი ასინქრონულ ოპერაციას სნქრონულის სტილში წერს. ერორ ჰენდლინგი კი სინქრონული **try..catch** ბლოკის საშუალებით ხდება:

1. exports.getPosts = async (req, res, next) => {
2. const currentPage = req.query.page || 1;
3. const perPage = 2;
4. try {
5. const totalItems = await Post.find().countDocuments();
6. const posts = await Post.find()
7. .skip((currentPage - 1) \* perPage)
8. .limit(perPage);
10. res.status(200).json({
11. message: "Fetched posts successfully",
12. posts: posts,
13. totalItems: totalItems,
14. });
15. } catch (err) {
16. if (!err.statusCode) {
17. err.statusCode = 500;
18. }
19. next(err);
20. }
21. };

**26. Understanding Async Await in Node.js**

401. Top-level "await"

**Top-Level Await:** იგი საშუალებას გვაძევს **await**ქივორდი ასინქრონული ფუნქციის გარეთაც გამოვიყენოთ, ანუ:

1. await smth...

**26. Understanding Async Await in Node.js**

403. Wrap Up

**mongoose**- მასში ჩაშენებულ მეთოდებში **Promise**-ის ნაცვლად **Object-like Promise**-ს აბრუნებს.

თუ გვსურს, რომ რეალური Promise მივიღოთ, თითოეულ მეთოდს ბოლოში უნდა მივაბათ **.exec()**:

1. const totalItems = await Post.find().countDocuments().exec()

**27. Understanding Websockets & Socket.io**

408. Setting Up Socket.io on the Server

**WebSocket**პროტოკოლი  **HTTP**პროტოკოლზე აგებული უფრო განვითარებული ვერსიაა, რომლით სარგებლობის საშუალებასაც ბევრნაირი პაკეჯი გვთავაზობს.

დანარჩენი ფოტოზე...

ყველაზე პოპულარული პაკეჯია **Socket.IO**

იგი უნდა დაემატოს როგორც ფრონტის, ისე ბექის მხარესაც.

*გაგრძელდება...*

**27. Understanding Websockets & Socket.io**

408. Setting Up Socket.io on the Server

**Back-End:**

ვაინსტალირებთ:npm install --save socket.io

შემდეგ კი, რამდენადაც ვებსოკეტის პროტოკოლი **http**-ზეა დამყარებული, მას წვდომა უნდა ჰქონდეს ჩვენს http რექვესტებზე.

ასევე, ჩვენი ფუნქცია **.listen()** რეალურად აბრუნებს სერვერს, როგორც ასეთი, და იგი შეგვიძლია მივიჩნიოთ თავად http კომუნიკაციად (რამდენადაც **HTTP**მოდულიდან მოდის საბოლოოდ ეგ სერვერი).

შემდეგ კი დავაიმპორტებთ**socket.io**-ს, რომელიც აბრუნებს ფუნქციას და ზედვე, იმპორტზევე, გამოვიძახებთ ფუნქციას და არგუმენტად ჩავაწვდით ჩვენს სერვერს, რომელიც ცვლადში შევინახეთ.

ასევე, მნიშვნელოვანია, რომ სერვერის მერე, მეორე არგუმენტად, ფუნქციას ჩავაწოდოთ **cors**პოლიტიკის ობიექტი, სადაც განვსაზღვრავთ საიდან შეიძლება რექვესტის მიღება და რომელი მეთოდები შეიძლება იქნას გამოყენებული.

შემდეგ თავად ამ ცვლადს, სადაც ეს გამოძახებები ინახება, ან აქვს მრავალი მეთოდი, მათ შორისაა **.on()**, რომელიც იღებს, როგორც ოფშენალ, ისე წინასწარ განსაზღვრული არგუმენტებს.

*გაგრძელდება...კოდი შემდეგში...*

**27. Understanding Websockets & Socket.io**

408. Setting Up Socket.io on the Server

**Back-End** კოდის მაგალითი:

**გაგრძელება:**

1. const server = app.listen(8080, () => {
2. console.log("Server connected on 8080 port!");
3. });
4. const io = require('socket.io')(server, {
5. cors: {
6. origin: "http://localhost:3000",
7. methods: ["GET", "POST"]
8. }
9. });
10. io.on('connection', socket => {
11. console.log('Client Connected!')
12. })

**27. Understanding Websockets & Socket.io**

409. Establishing a Connection From the Client

**Front-End:**

აინსტალირებთ:npm install --save socket.io-client

(ან **CMD**-ით, ანუ სკრიპტის გალინკვით, რომელიც მის დოკუმეტნაციაშია)

შემდეგ ვაიმპორტებთ:

1. import openSocket from 'socket.io-client'

შემდეგ კი, მის გასაშვებად:

1. openSocket('http://localhost:8080')

**27. Understanding Websockets & Socket.io**

411. Sharing the IO Instance Across Files

სასურველია **Socket.io** ერთ ფაილში დავადეკლარიროთ, რათა ყველა სხვა ფაილიდან ხელმისაწვდომი იყოს და დაქონექთებაც შესაბამისად მაგ ფაილიდანვე გავაკეთოთ.

აღნიშნული ფაილი დააექსპორტებს პირველ რიგში თავად სერვერთან დაქონექთებას, რომელსაც ქონექშენი გამოიძახებს, და ასევე IO მეთოდებს, რომელსაც სხვა კომპონენტები გამოიძახებენ

socket.js:

1. let io;
3. module.exports = {
4. init: httpServer => {
5. io = require('socket.io')(httpServer, {
6. cors: {
7. origin: "http://localhost:3000",
8. methods: ["GET", "POST"]
9. }
10. })
11. return io
12. },
13. getIO: () => {
14. if(!io) {
15. throw new Error('Socket.io not initialized!')
16. }
17. return io;
18. }
19. }

საწყისი ქონექშენისას გამოძახება **init**მეთოდის (app.js):

1. const server = app.listen(8080, () => {
2. console.log("Server connected on 8080 port!");
3. });
5. const io = require('./socket').init(server);
7. io.on('connection', socket => {
8. console.log('Client Connected!')
9. })

**27. Understanding Websockets & Socket.io**

412. Synchronizing POST Additions

რამდენადაც გლობალურად დექსპორტებული **io** ობიექტი, რომელიც ფუნქცია **getIO**-ში ინახება, შეგვიძლია ნებისმიერგან დავაიმპორტოთ. ასევე მასში შენახულ **socket.io-ს** ფუნქციებითაც ვისარგებლებთ.

მაგალითად, ბექიდან გავგზავნით კონკრეტულ ინფოს კონკრეტული ქონექშენის სახელით, რომლისთვისაც**.emit()**მეთოდს გამოვიყენებთ, პირველი არგუმენტია ქონექშენის სახელი (ჩემზეა), მეორე კი დეითა, მაგალითად ობიექტი:

1. const io = require('../socket');
2. io.getIO().emit('posts', { action: 'create', post: post })

ფრონტის მხარის დაქონექთებას კი ცვლადში შევინახავ და ამ ცვლადში არსებულ მეთოდებს მივწვდები, მათ შორისაა **.on()**, რომლის პირველი არგუმენტიცაა ქონექშენ სტრინგი, რომელსაც უნდა მოუსმინოს და მეორე ქოლბაქ ფუნქცია, სადაც დეითა ინახება:

1. const socket = openSocket('http://localhost:8080');
2. socket.on('posts', data => {
3. if(data.action === 'create') {
4. this.addPost(data.post);
5. }
6. })

**27. Understanding Websockets & Socket.io**

417. Wrap Up

**Useful resources:**

* **Socket.io** Official Docs: https://socket.io/get-started/chat/
* Alternative **Websocket**Library: https://www.npmjs.com/package/express-ws

**28. Working with GraphQL**

421. Understanding the Setup & Writing our First Query

***GraphQL****-ზე ინფო ძირითადად ფოტოებზეა...*

მის ბექში გამოსაყენებლად დაგვჭირდება ორი პაკეჯი, რომლებსაც შემდეგნაირად დავაინსტალირებთ: npm install --save graphql express-graphql

მისი გამოყენება შესაძლებელია სხვადასხვა ენაში, მათ შორის, ჯავასკრიპტშიც, ასევე სხვადასხვა ფრეიმვორკშიც, რომელზეც ინფოს შემიძლია მის დოკუმენტაციაში გავეცნო.

**graphql**პაკეჯი გვეხმარება სქიმას შექმნაშ, ანუ როგორ უნდა გამოიყურებოდეს სქიმა.

**express-graphql** კი კონკრეტულ ფუნქციებს ემსახურება, ანუ იყენებს, რომლებიც კავშირში არიან **graphql**-ის სქიმასთან.

საწყისისთვის შეგვიძლია როუტების გაუქმება და ცალკე **graphql**-ისთვის ფოლდერის შექმნა, სადაც ორი ფაილი იქნება: **schema.js** და **resolver.js**.

შემდეგ კი **app.js**-ში მოხდება საჭირო ფუნქციონალის შემოტანა.

**28. Working with GraphQL**

421. Understanding the Setup & Writing our First Query

**schema.js**-ში ვაიმპორტებთ **graphql**-ს, სადაც სქიმა უნდა შევქმნათ. შესაბამისად, **buildSchema**უნდა დავაიმპორტოთ, შემდეგ კი დავაექსპორტოთ აღნიშნული, როგორც ფუნქცია, რომელიც არგუმენტად სქიმას სტრუქტურას იღებს, სადაც ასევე დეკლარირდება დეითას ტიპები.

*მას თავისებური სინტაქსი აქვს, მაგალითად მძიმეები არ გამოიყენება ტიპებში...*

თუ **!**- ნიშანია გამოყენებული ტიპის შემდეგ, ნიშნავს რომ **required**-ს ხდი კონკრეტულ ტიპს.

1. const { buildSchema } = require("graphql");
3. module.exports = buildSchema(`
4. type TestData {
5. text: String!
6. views: Int!
7. }
8. type RootQuery {
9. hello: TestData!
10. }
11. schema {
12. query: RootQuery
13. }
14. `);

**28. Working with GraphQL**

421. Understanding the Setup & Writing our First Query

ქიურიებს (მოთხოვნებს) ასევე ჭირდებათ რესოლვერი, რომელიც მათში განხორციელდება, შესაბამისად**resolver.js**-ში იწერება ზუსტად ეგ რესოლვერის ფუნქცია, რომელიც ექსპორტდება. აღნიშნული რესოლვერი შეესაბამება წინაზე შექმნილ სქიმას:

1. module.exports = {
2. hello () {
3. return {
4. text: 'Hello World!',
5. views: 12345
6. }
7. }
8. }

**28. Working with GraphQL**

421. Understanding the Setup & Writing our First Query

ამ ორის დაკავშირება და საჯაროდ გატანა უკვე მოხდება **app.js**-ის დონეზე, სადაც გამოვიყენებთ, ანუ დავაიმპორტებთ, **express-graphql**-ის პაკეჯს და ასევე ჩვენს სქიმასა და რესოლვერს.

შემდეგ კი, დეფაულტ ერორ ჰენდლინგამდე, **app.use()**-ში პირველ არგუმენტად სტანდარტულად ჩავაწვდით **path**-ს **'/graphql'**, ხოლო მეორე არუგმენტად მიდის ჩვენი **graphqlHTTP**ფუნქცია (რაც დავაიმპორტეთ), რომელიც არგუმენტად იღებს ობიექტს და ასევე ორ ფროფერთის: **schema**და **rootValue**, სადაც ჩვენს დაიმპორტებულ სქიმასა და რესოლვერს გადავცემთ.

**app.js:**

1. const { graphqlHTTP } = require("express-graphql");
2. const graphqlSchema = require('./graphql/schema');
3. const graphqlResolver = require('./graphql/resolvers');
5. app.use('/graphql', graphqlHTTP({
6. schema: graphqlSchema,
7. rootValue: graphqlResolver
8. }))

*გაგრძელდება...*

**28. Working with GraphQL**

421. Understanding the Setup & Writing our First Query

შემდეგ კი, რევესტის გაგზავნისას, **json**დეითას გავუგზავნით **POST**რექვესტით მითითებულ **path**-ზე, სადაც **key**იქნება **"query"**, ხოლო **value**იქნება ობიექტი, რომელიც ჩვენი სტრუქტურის შესაბამისად იქნება შემდეგნაირი:

1. {
2. "query": "{ hello { text views } }"
3. }

აქ, თუ მარტო **text**-ს მივუთითებთ, მარტო მაგ დეითას მოიტანს.

**28. Working with GraphQL**

422. Defining a Mutation Schema

ჩვენი შემთხვევისთვის დაგვჭირდება მაგალითად რეგისტრაციის ქიმას შექმნა. სტანდარტულად შევქმნით, უბრალოდ აქ ახალი იქნება: **input**ტიპი, რომელიც გამოიყენება ინფუთის დეითას ტიპისთვის,**()** - ფუნქციის ტიპი, რომელიც განსაზღვრავს რესოლვერში ჩასაწოდებელი ინფუთის ტიპს, **ID**ტიპი, რომელიც დეითას უნიკალურობას უსვამს ხაზს.

*კოდი შემდეგში...*

**28. Working with GraphQL**

422. Defining a Mutation Schema

1. const { buildSchema } = require("graphql");
3. module.exports = buildSchema(`
4. type Post {
5. \_id: ID!
6. title: String!
7. content: String!
8. imageUrl: String!
9. creator: User!
10. createdAt: String!
11. updatedAt: String!
12. }
14. type User {
15. \_id: ID!
16. name: String!
17. email: String!
18. password: String
19. status: String!
20. posts: [Post]
21. }
23. input UserInputData {
24. email: String!
25. name: String!
26. password: String!
27. }
29. type RootMutation {
30. createUser(userInput: UserInputData): User!
31. }
33. schema {
34. mutation: RootMutation
35. }
36. `);

**28. Working with GraphQL**

423. Adding a Mutation Resolver & GraphiQL

მანამ, სანამ **resolver**-ს ფროდაქშენზე გავუშვებდეთ, შეგვიძლია **app.use()**-ში, სადაც გრაფის სეტაპი გვაქვს, ობიექტს ჩავამატოთ ახალი ფროფერთი - **graphiql**(**GraphicalQL**), რომელსაც ვაქცევთ **true**-დ, იგი საშუალებას მოგვცემს როუტში ჩვნს მიერ მითითებული path (**/graphql**) შევიყვანოთ ბრაუზერიდან და ისეგავტესტოთ კოდი:

1. app.use('/graphql', graphqlHTTP({
2. schema: graphqlSchema,
3. rootValue: graphqlResolver,
4. graphiql: true
5. }))
7. // შემდეგ ბრაუზერში, მაგ: http://localhost:8080/graphql

ასევე საჭიროა სქიმაში ჩავამატოთ **RootQuery**და **query**:

1. type RootQuery {
2. hello: String
3. }
4. schema {
5. query: RootQuery
6. }

**28. Working with GraphQL**

423. Adding a Mutation Resolver & GraphiQL

შემდეგ კი, **GraphQL**ისეთ ინტერფეისს გამოიტანს, რომელშიც შეგვიძლია ფრონტის იმიტაცია გავაკეთოთ და ისე გავგზავნოთ ბრძანებები, მაგ:

1. mutation {
2. createUser(userInput:{email:"test@test.com", name:"Giorgi Charkviani", password:"giorgi"}) {
3. \_id
4. email
5. password
6. }
7. }

**28. Working with GraphQL**

423. Adding a Mutation Resolver & GraphiQL

რესოლვერი ასე გამოიყურება:

1. const User = require('../models/user');
2. const bcrypt = require("bcryptjs");
4. module.exports = {
5. createUser: async function({ userInput }, req) {
6. const existingUser = await User.findOne({email: userInput.email})
7. if(existingUser) {
8. const error = new Error('User exists already!')
9. throw error;
10. }
11. const hashedPw = await bcrypt.hash(userInput.password, 12)
12. const user = new User({
13. email: userInput.email,
14. name: userInput.name,
15. password: hashedPw
16. })
17. const createdUser = await user.save();
18. return {...createdUser.\_doc, \_id: createdUser.\_id.toString()}
19. // ასე იმიტომ, რომ \_id-ზე გადაწერა მოხდეს, რადგან გრაფი String-ს ელის
20. }
21. }

თუ სტანდარტულ **then()** ჩეინს გამოვიყენებდი, **return**უნდა გამეკეთებინა User.findOne().then()-ისთვის, რადგან  GraphQL არ დაელოდებოდა მას. **async..await**-ს არ ჭირდება ეს

**28. Working with GraphQL**

424. Adding Input Validation

**GraphQL**-ში წინა ვალიდატორს ვერ გამოვიყენებთ, ამისთვის დაგვჭირდება ახალი ვალიდატორი, რომელსაც ასევე წინა ვალიდატორიც იყენებდა, მას კი შემდეგნაირად დავაინსტალირებთ:npm install --save validator

შემდეგ მას**resolvers.js**-ში დავაიმპორტებთ და ჩვენს რესოლვერის ფუნქციაში გამოვიყენებთ:

1. const validator = require("validator");
3. createUser: async function ({ userInput }, req) {
4. const errors = [];
5. if (!validator.isEmail(userInput.email)) {
6. errors.push({ message: "Email is invalid." });
7. }
9. if (
10. validator.isEmpty(userInput.password) ||
11. !validator.isLength(userInput.password, { min: 5 })
12. ) {
13. errors.push({message: 'Password too short!'})
14. }
16. if(errors.length > 0) {
17. const error = new Error('Invalid input.')
18. throw error;
19. }
20. ...
21. })

**28. Working with GraphQL**

425. Handling Errors

თუ რაიმე ერორი მოხდება, GraphQL თავის სტილში გამოგიგზავნის ერორს. თუ გვსურს ერორის ობიექტის დაკონფიგურირება, ამისთვის app.use()-ის დონეზე უნდა ჩავაწოდოთ კიდევ ერთი ფროფერთი - customFormatErrorFn(err) - რომელშიც შევძლებთ მის დაკონფიგურირებას. იგი ასევე მოიცავს ჩაშენებულ ერორს, რომელიც ტექნიკური ერორისას აქტიურდება და აღნიშნულიც უნდა გავითვალისწინოთ (originalError). შემდეგნაირი სახე ექნება ჩვენს კონფიგურაციას:

resolver.js:

1. if(errors.length > 0) {
2. const error = new Error('Invalid input.')
3. error.data = errors;
4. error.code = 422;
5. throw error;
6. }

app.js:

1. app.use('/graphql', graphqlHTTP({
2. schema: graphqlSchema,
3. rootValue: graphqlResolver,
4. graphiql: true,
5. customFormatErrorFn(err) {
6. if(!err.originalError) {
7. return err;
8. }
9. const data = err.originalError.data;
10. const message = err.message || 'An error occured.'
11. const code = err.originalError.code || 500
12. return {
13. message: message,
14. status: code,
15. data: data
16. }
17. }
18. }))

**28. Working with GraphQL**

426. Connecting the Frontend to the GraphQL API

კიდევ ერთი ეჯასტმენტი გვჭირდება, რათა **CORS**პრობლემა არ წარმოიშვას:

როგორც უკვე ვიცით, ბრაუზერი ავტომატურად, რექვესტის გაგზავნამდე, გზავნის **'OPTIONS'** ტიპის რექვესტს, რომელსაც **GraphQL**-ის არ იღებს და შესაბამისად დროპავს. ამი გამოსასწორებლად, იქ, სადაც ჰედერები გვაქვს, უნდა შევამოწმოთ, თუ რექვესტი **'OPTIONS'** ტიპის არის, პასუხად გავუგზავნოთ **200**სტატუსი **.sendStatus()** მეთოდის გამოყენებით.

შედეგად, რექვესტი **GraphQL**-ამდე ვეღარ მიაღწევს და არც ბრაუზერი აღარ დაბლოკავს.

ეს კოდი უნდა ჩავწეროთ მანამ სანამ **next()** გაეშვება, ანუ **next()**-მდე:

1. if(req.method === 'OPTIONS') {
2. return res.sendStatus(200)
3. } // GraphQL-ამდე რომ არ მივუშვათ 'OPTIONS' რექვესტი
4. next();

**28. Working with GraphQL**

426. Connecting the Frontend to the GraphQL API

**Front**-ის მხრიდან კი არ უნდა დაგვავიწყდეს, რომ რექვესტებს ვაგზავნით მხოლოდ ერთ ენდპოინტზე - **/graphql** - როგორც გავმართეთ, და ასევე **POST**ტიპის უნდა იყოს ყველა რექვესტი.

აი, დეითა კი, ანუ რასაც ვაგზავნით და რასაც მივიღებთ, **GraphQL**-ის სინტაქსით უდა დავწეროთ, როგორც სატესტო გარემოში დავწერე, მსგავსად და ეგ გავასტრინგიფაიოთ მერე:

**reactjs:**

1. const graphqlQuery = {
2. query: `
3. mutation {
4. createUser(userInput:{email:"${authData.signupForm.email.value}", name:"${authData.signupForm.name.value}", password:"${authData.signupForm.password.value}"}) {
5. \_id
6. email
7. }
8. }
9. `,
10. };
12. fetch("http://localhost:8080/graphql", {
13. method: "POST",
14. headers: { "Content-Type": "application/json",},
15. body: JSON.stringify(graphqlQuery),
16. })
17. .then((res) => {
18. return res.json();
19. })