**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. }