IZGRADNJA SUSTAVA ZA E-UČENJE TEMELJENOG NA NODE.JS PLATFORMI

Drnasin, Luigi

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Stručni diplomski studij Elektrotehnika

LUIGI DRNASIN

Z A V R Š N I R A D

IZGRADNJA SUSTAVA ZA E-UČENJE TEMELJENOG NA NODE.JS PLATFORMI

Split, travanj 2024.

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Stručni diplomski studij Elektrotehnika

Predmet: Sigurnost mreža i usluga

Z A V R Š N I R A D

Kandidat: Luigi Drnasin

Naslov rada: Izgradnja sustava za e-učenje temeljenog na Node.js platformi

Mentor: Tonko Kovačević

Split, travanj 2024.

SADRŽAJ

Sažetak 1
Sustav za e-učenje ..... **Error! Bookmark not defined.**
Summary ..... **Error! Bookmark not defined.**
E-learning system ..... **Error! Bookmark not defined.**
1. UVOD 2
2. TEHNOLOGIJE 4
2.1. JavaScript 4
2.1.1. Pokretanje 4
2.2. Node, Express 5
2.3. Postgres, Prisma 5
2.4. Typescript 7
3. REST API 8
4. KLIJENT-SERVER ARHITEKTURI 11
4.1. Arhitektura 11
4.2. Kolačići 13
4.3. CORS 15
4.4. Status kodovi 17
5. AUTENTIFIKACIJA I AUTORIZACIJA 20
5.1. JWT 20
5.2. Autentifikacija 22
5.3. Autorizacija 30
5.3.1. Implementacija autorizacije 33
6. ZAKLJUČAK 38
LITERATURA 39
POPIS SLIKA 40

Izgradnja sustava za e-učenje temeljenog na Node.js platformi

Sažetak:

Projekti zadatak je bio napraviti web aplikaciju sa fokusom na sigurnost, glavna problematika rada je autentifikacija i autorizacija. Autentifikacija će se riješiti na način da će se generirati token nakon login i taj će se token slati svakim sljedećim *request*-om te će se na backendu provijeriti njegova validnost. Osim sigurnosti napravljena je REST API arhitektura, te su izvršene sve CRUD operacije. Napravljena aplikacija je u potpunosti skalabilna te se logički principi sigurnosti mogu primjeniti na bilo koju aplikaciju koja sadrži registraciju i login korisnika.

Ključne riječi: autetifikacija, autorizacija, REST, token

Building an e-learning system based on the Node.js platform

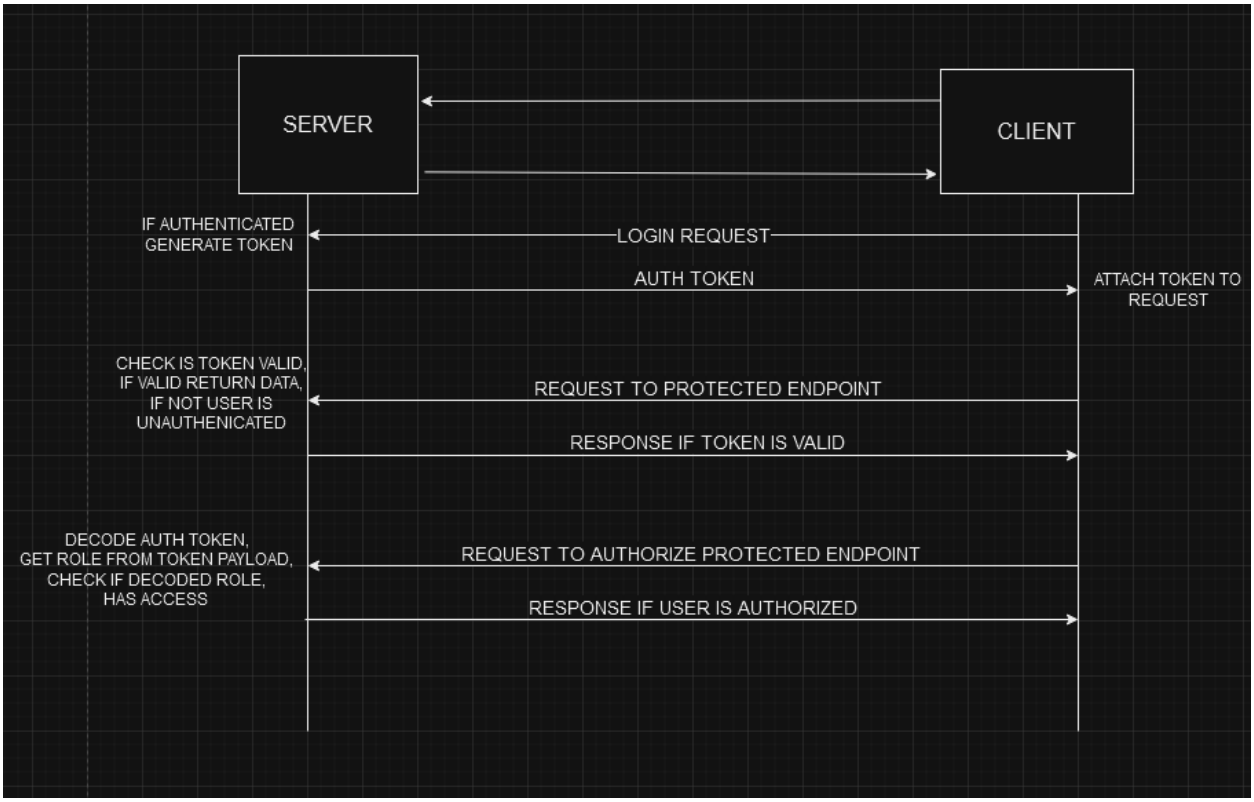
Abstact:

Project task was to build full stack web application with a focus on security, i.e. authentication and authorization. Authentication will be solved as follows, after the user logs in, a token will be generated on server and that token will be attached to all future requests, and the token will be validated on server. In addition to security, REST API architecture is built with all CRUD operations. Application is fully scalable and security patterns presented can be applied to any other application that contains registration and login.

Key words: authentication, authorization, REST, token

1. UVOD

Iz samog naslova diplomskog rada vidljivo je da se radi o web aplikaciji odnosno web stranici. Svaka web stranica se sastoji od dva glavna dijela, a to su *backend* i *frontend*. Pod *backend*-om se smatra server u kojem je glavna logika i baza koja nam služi za spremanje podataka koji moraju biti trajno sačuvani dok je *frontend* dio stranice koji nema trajnu memoriju te je vidljiv korisniku i na taj način se izvršava interakcija korisnika sa aplikacijom. U diplomskom radu glavni fokus će biti na funkcioniranju i principima *backend*-a. Na *backend*-u i *frontend*-u će se koristiti programski jezik JavaScript, koja se pokreće u web pregledniku, a na backendu ćemo koristiti nodejs koji je "run time environment" za JavaScript. Drugim riječima nodejs omogućava da se JavaScript pokreće na serveru, odnosno bilo kojoj mašini koja sadrži nodejs što inače nije moguće. U daljnem radu će se opisati sve tehnologije korištene pri izradi web stranice gdje će se veći fokus posvetiti tehnologijama backenda. Nakon toga će se objasniti što je REST (engl. *Representational state transfer*) API (engl. *Application programming interface*), i koje kriterije server mora zadovoljit da ga se naziva REST. Pojasnit će se detaljnije arhitekturu servera te elementarne stvari kao što su: CORS (engl. *Cross-origin resource sharing*), kolačići i status kodovi. Nakon toga će se ući u samu logiku aplikacije, te će se kroz primjere prikazati što je autentifikacija i autorizacija te kako je ona riješena pri izradi aplikacije. Autentifikacije će se riješevati preko JWT (engl. *JSON web token*), koji će se generirati nakon što se korisnik logira u sustav, dok će se autorizacija riješavati na sličan način gdje će se rola korisnika također zapisati u JWT te će se na backendu vršiti provjera prava pristupa. Svaki korisnik će moći imati jednu od 3 predefinirane role: "admin", "teacher", "student". Osim endpointa za autentifikaciju (login, register, logout) server će posjedovat dva api endpointa: "students" i "books" o kojima će se ući u detalje malo kasnije u radu. Na kraju će se rad aplikacije opisati u cijelosti odnosno što se sve dešava u pozadini tijekom logina i boravka korisnika u aplikaciji. Na slici 1.1. se vidi pojednostavljeni princip rada autentifikacije i autorizacije u aplikaciji.



Slika 1.1. Autentifikacija i autorizacija

2. TEHNOLOGIJE

2.1. JavaScript

Prema literaturi[1] JavaScript je "*multi-paradigm*" jezik, što zapravo znači da se može pisati odnosno konstruirati na više načina. Postoje tri osnovne paradigme koje se koriste za pisanje i arhitekturiranje koda, a to su: proceduralna, objektno-orjentirana (OOP), i funkcionalna (FP). Ne postoji ispravna paradigma, to su samo orijentacije i upute prema kojima se piše kod. Neki programski jezici su strogo orijentirani prema jednoj paradigmi (C je proceduralan, dok su Java i C++ OOP). JavaScript se može pisati u svim paradigmama. Također JavaScript je *backward compatible* što znači da sav kod napisan u JavaScript viječno raditi bezobzira što su uvedeni noviteti u jeziku. JavaScript je dinamičan što znači da varijabla na početku može biti tipa string, a kasnije tipa number.

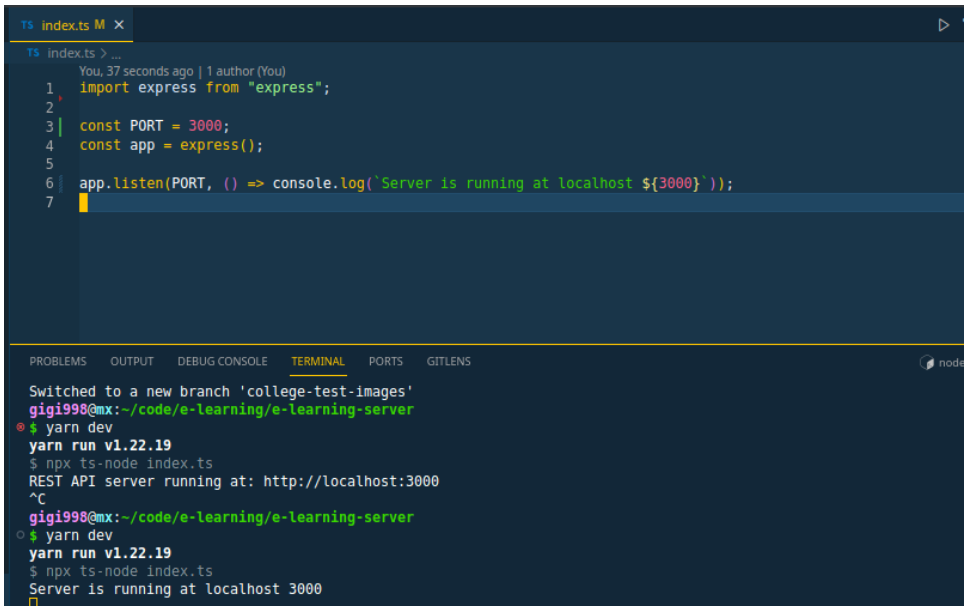
2.1.1. Pokretanje

Svaki programski jezik da bi bio razumljiv računalu mora se pretvoriti u bajt-kod (engl. *Bytecode*) te nakon toga u *machine-code*. *Machine-code* je oblik u kojem računala funkcioniraju. Sama pretvorba iz napisanog koda u *machine-code* se može vršiti na dva načina, to su kompajler i interpreter. Razlika je što kompajler prođe kroz cijeli kod te krerira novi file sa kompajliranim kodom, proces kompajliranja oduzima nešto vremena, ali nakon toga je program brz, dok sa druge strane interpreter obrađuje kod liniju po liniju te ga tako pretvara u *machine-code*, interpreter je inicijalno brži zato što izvođenje kreće odma, ali lako su mogući problemi sa optimizacijom ili vremenski dugim petljama pogotovo kod velikih aplikacija. JavaScript kao jezik koristi oboje, takozvani *just in time compailer*. Raspored što će se desiti prvo, a što drugo ovisi o *engine*. Svaki web preglednik sadrži u sebi JavaScript *engine*, najpoznatiji je Google-ov koji se naziva V8. Prvi korak u pretvaranju JavaScript u *machine-code* je parsiranje i pretvaranje koda u tokene sa značenjem. Tokeni čine AST (*Abstract Syntax Tree*). Nakon toga AST je prosljeđen interpreteru koji generira ne optimizirani *machine-code* čije izvšavanje može početi odma, nakon toga takozvani *Profiler* analizira kod i traži gdje se može izvršiti optimizacija, nakon pronalaska dijela koda koji se može optimizirat taj dio šalje kompajleru. Nakon kompajla novo kreirani kod zamjenjuje ne optimizirani dio koda generiran od strane interpretera, tako da *Profiler* stalno pregledava kod i kompajlira ga te ga čini bržim.

2.2. Node, Express

Kao što je već spomenuto Node je JavaScript *runtime environment* što omogućuje da se JavaScript pokreće na serveru, a ne u web pregledniku kako je objašnjeno u prethodnoj sekciji. Nodejs koristi već spomenuti V8 *engine* od Googla. U Node-u su nam dostupne funkcije i paketi koji nam inače nisu dostupne kad se JavaScript pokreće u web pregledniku, neke od njih su: može se saznati u kojem se folderu nalazimo, na kojoj smo mašini, možemo pisati i trajno spremiti podatke, te čitati podatke.

Express je Nodejs *framework* koji u sebi ima alate koji nam pomažu pri organiziranju i kreiranju server aplikacija. Kreiranje Express servera je vrlo jednostavno što je prikazano na slici 2.1.



Slika 2.1. Kreiranje servera

2.3. Postgres, Prisma

Podatci se spremaju u SQL bazu koja koristi PostgresSQL za upravljanje i manipuliranje bazom. Relacijske baze podataka se sastoje od tablica koje u sebi imaju retke i stupce. Povezivanje tablica se vrši na način da se kreiraju relacije pomoću primarnog i stranog ključa. Svaka relacijska baza

podržava *joint* što je način da povežemo i povučemo željene podatke iz više tablica koje su u relaciji.

Prisma je alat koji nam omogućava da vršimo upite na bazu vrlo jednostavno. Prisma u pozadini konvertira naš kod u SQL te vrši upite na bazu. Na slici 2.2. se vidi inicijalno postavljanje prisma šeme te kreiranje relacije jedan na više što znači da jedan student može posuditi više knjiga, dok svaka knjiga pripada samo jednom studentu. U modelu Book id je primarni ključ, a studentId je strani ključ pomoću kojeg se vrši relacija sa tablicom Student.

Slika 2.2. Prisma schema

Nakon kreiranja Prisma šeme potrebno je pokrenuti dvije naredbe:

yarn prisma migra dev

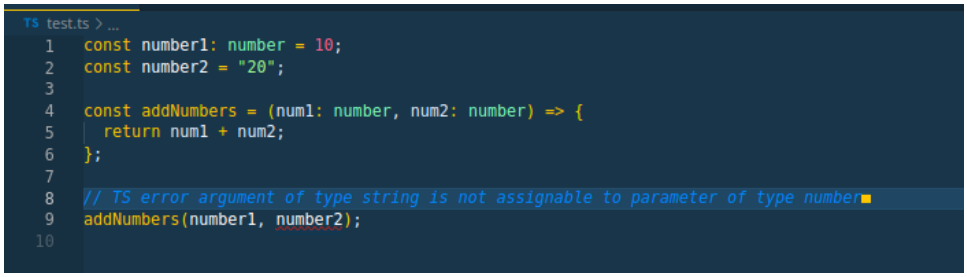
yarn prisma generate

Prva naredba migrira kreiranu šemu u bazu, tako da nakon izvršavanja naredbe u bazi će se nalaziti tablice “Student” i “Book”.

Druga naredba generira Prisma klijent pomoću kojeg vršimo upite na bazu. Tako da nakon svake promjene u šemi potrebno je pokrenuti *prisma migrate* i *prisma generate*.

2.4. Typescript

Typescript je prošireni JavaScript, pomoću njega je moguće definirati tipove podataka u JavaScriptu što nam omogućuje otkrivanje erora u ranoj fazi razvoja aplikacije. Tako da Typescript čini JavaScript kod statičan, a ne dinamičan kakva je JavaScript. Na slici 2.3. su kreirane dvije varijable, i funkcija korištenjem Typescripta. Drugoj varijabli nije definiran tip.



Slika 2.3. Typescript eror

Prvi pozivanju kreirane funkcije Typescript javlja eror zato što oba parametra funkcije očekuju tip broj (eng. *number*), a varijabla “number2” je tipa string.

3. REST API

API je sučelje koje služi za komuniciranje dva računala na internetu. API se također može opisati kao dogovor između korisnika i servera u svrhu lakše komunikacije. REST je posebna vrsta API-a koja koristi definiranu REST arhitekturu i pravila te omogućuje komunikaciju sa REST servisima. REST nije standard ili protocol. Mnogi koriste REST API arhitekutru zbog njegove jednostavnosti i skalabilnosti. Iz literature [2] REST principi su:

- *UNIFORM INTERFACE*

Svaki *request* koji želi pristupiti istom *resource*(podatcima) treba izgledati jednako bez obzira odakle dolazi. REST mora omogućiti da svaki zasebni podatak pripada samo jednom URI (engl. *Unique resource identifier*).

- *STATELESS*

Što znači da svaki *request* sa klijenta mora sadržavati sve potrebne podatke kroz *headers, body, params* i *cookies* i da ga server može razumijeti i obraditi *request*. Server ne treba održavati nikakav state ili sesiju. Server ne sadrži nikakve podatke o klijentu.

- *CACHABLE*

Response mora sadržavati status da li se može predmemorirati (eng. *cachable*) ili ne. Ako je *response* predmemoriran to znači da će sljedeći *request* koristiti dohvatiti predmemorirane podatke.

- KLIJENT SERVER ARHITEKTURA

Klijent i server moraju biti podijeljeni, klijent je zadužen za slanje *requesta* na određeni *resource* dok server sadržava *resource*, vraća *response* i sprema trajno podatke. Servera ne zanima nikakav UI (engl. *User interface*) ili state korisnika, dok klienta ne zna ništa o logici na backendu.

- VIŠE SLOJEVA

REST API se može kreirati u više slojeva, server može biti u sloju A, baza u sloju B, te pritom klijent ne treba znati jeli povezan na krajnji server na zadnjem sloju ili na neki drugi sloj u arhitekturi.

- KOD NA ZAHTJEV

U većini slučajeva *response* će sadržavati podatke u obliku JSON-a (engl. *Javascrpit object notation*), ali je također moguće vratiti programski kod koji se može izvršiti.

Također za kreiranje ispravnog REST API moraju se pratiti određena pravila:

- REST *endpoint* se bazira na *resource*-u ili na imenici u množini,a ne na akciji koju izvršava. Na slici 3.1. se vide dvije api route koje će se koristiti u aplikaciji.

```
30
31  // API ROUTES
32  app.use("/api/students", require("./routes/api/students"));
33  app.use("/api/books", require("./routes/api/books"));      You, 3 months ag
34
35  app.listen(3000, () =>
36    console.log("REST API server running at: http://localhost:3000")
37  );
38
```

Slika 3.1. API rute

- Akcija koju REST *endpoint* treba izvršiti se definira prema HTTP metodama: GET, PUT, POST, DELETE.

- Aplikacija mora biti organizirana u rute te svaka ruta u endpointe te uz pomoć HTTP metode se definira akcija koju je potrebno izvšit na tom *endpoint*-u. Na slici 3.2. se vidi API ruta "books" sa dva *endpoint*-a koji ima 3 akcije na prvom *endpoint*-u, GET dohvaćanje svih podataka sa tog *endpoint*-a, POST kreiranje novih podataka i DELETE brisanje podataka.

```
router.route("/").get(getAllBooks).post(addNewBook).delete(deleteBook);

router.route("/:id").get(getSingleBook);

module.exports = router;
```

Slika 3.2. Endpoint books

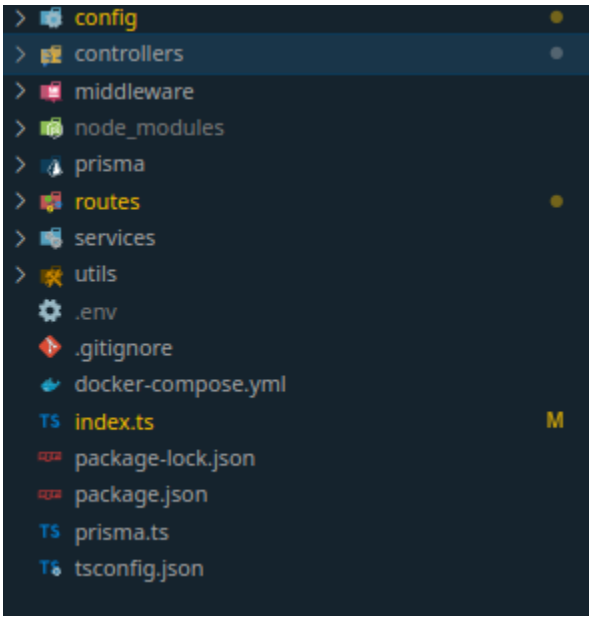
REST API nam nudi veliku fleksibilnost za širenje aplikacije na vrlo jednostavan način, uz potpunu ne ovisnost o klijentu donosi brzinu i skalabilnost.

4. KLIJENT-SERVER ARHITEKTURI

U ovom poglavlju će se opisati arhitektura i logika backend aplikacije u expressu, pojasnit će se ukratko što su cookies ili kolačići, zašto trebamo CORS, te koji su najkorišteniji status kodovi.

4.1. Arhitektura

Prilikom arhitekturiranja Express servera, aplikaciju smo podjelili na više logičkih dijelova vidljivo na slici 4.1.



Slika 4.1. Arhitektura servera

- Sve kreće od datoteke “index.ts” gdje se kreira server i definiraju rute.
- *Package-lock.json* i *package.json* su datoteke gdje su instalirani dodatni paketi, npr: Express, Prisma, Typescript itd.
- *Prisma.ts* datoteka nam služi za konfiguraciju Prisma klijenta.
- *Tsconfig.json* je Typescript konfiguracija.
- .env datoteka u kojem su spremljene tajne varijable
- *Docker-compose.yml* datoteka za konfiguraciju dockera

Osim spomenutih datoteka detaljnije će se opisati sljedeći folderi: “routes”, “controllers”, “services” i na kraju “middleware”.

Unutar *routes* foldera definiramo *endpoint*-e, postavljamo HTTP metodu, postavljamo *middleware* i pozivamo kontroler.

Middleware folder sadrži funkcije koje se najčešće pozivaju prije samog kotrolera ili čak prije cijele rute. Njihova zadaća je napraviti neku radnju prije nego što je *request* došao do kontrolera, najčešće ta radnja je provjera autentifikacije, autorizacije, validnosti itd. Moguće je pozvati neograničeni broj *middleware* funkcija prije samog kontrolera. Svaki *middleware* ima pristup *request* i *responsu* objektu i *next* funkciji. Tako da po potrebi *middleware* može iskoristiti podatke iz *requesta* te vršiti provjere, ako provjera koju je *middleware* izvršio je neuspješna, jednostavno se vrati *response* sa odgovorajučim eror status kodom, ako je pak provjera uspješna samo se pozove *next* funkcija koja nastavi sa daljnim izvršavanjem koda, što može biti sljedeći *middleware* ili kontroler. Na slici 4.2. je vidljiv *middleware* za validaciju *inputa*.

```
export const customValidator = (
  validate: Validators,
  body: string[]
) => {
  return function (
    req: Request,
    res: Response,
    next: NextFunction
  ) {
    for (const field of body) {
      if (!req[validate] || !req[validate][field]) {
        return res
          .status(403)
          .json({ message: `${field} required` });
      }
    }
    next();
  };
};
```

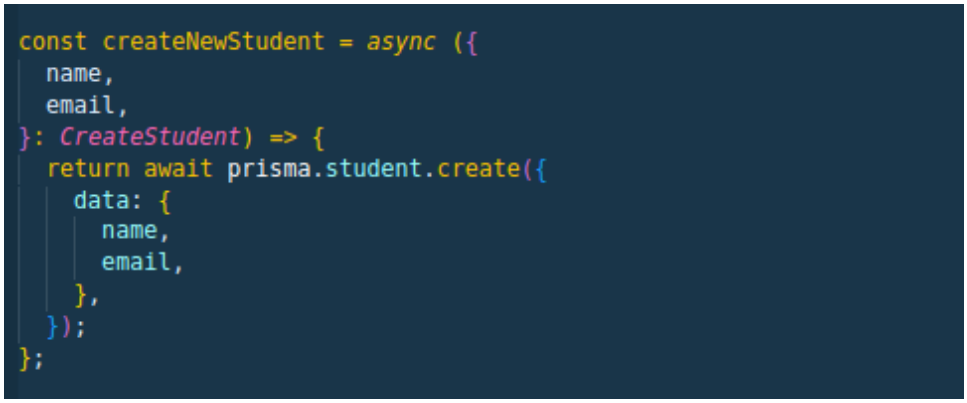
Slika 4.2. Middleware za validaciju

Kontroler je funkcija koja također ima pristup *request* i *response*-u objektu i *next* funkciji, ali njegova je svrha bitno drugačija, naime u kotroleru se riješava glavna logika *endpoint*-a. Kontroler razdvaja rutu od same logike. U njemu se poziva servis koji radi upit na bazu, te se zavisno o odgovoru servisa vraćaju podatci sa prikladnim status kodom. Na slici 4.3. je prikazan

“getAllStudents” kontroler čija je zadaća dohvaćanje svih studenata iz baze, te vraćanje istih u JSON formatu.

Slika 4.3. Students kontroler

I posljedni servis koji izvršava logiku aplikacije odnosno sam upit na bazu. Bitno je naglasiti da servis i kontroler nemaju nikakve povezanosti jedan s drugim, što znači da se kreirani servisi mogu primjeniti na bilo kojoj aplikaciji. Na slici 4.4. je vidljiv student servis koji kreira novog studenta.



Slika 4.4. Kreiranje studenta

4.2. Kolačići

Kolačići (engl. *cookies*) su male tekstualne datoteke koje generira server i šalje u korisnikov web preglednik. U aplikaciji u kolačiće će se spremiti token koji će služiti za autentifikaciju korisnika. Pri kreiranju kolačića moguće je definirati parametre, koristiti će se samo nekolicina nama bitnih:

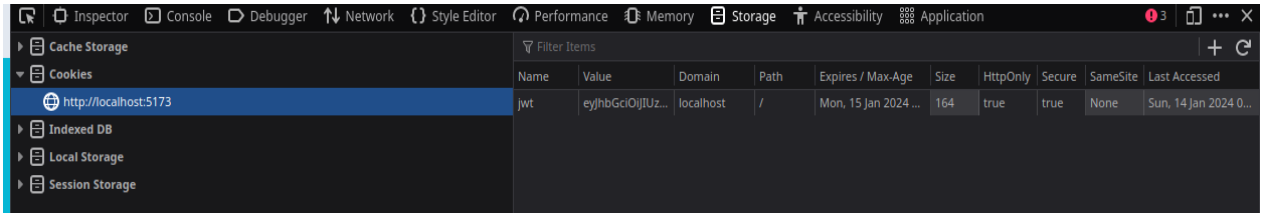
- *MaxAge* - vrijeme trajanja kolačića u milisekundama, nakon isteka postaje ne valjan
- *Secure* - ako je točno kolačić se šalje u enkriptiranoj formi samo preko HTTPS protokola
- *HttpOnly* – ako je točon kolačić nije dostupan JavaScriptu na frontendu, što je jako bitno zato što se u njega spremaju podatci o korisniku.

Na slici 4.5. se vidi kreiranje kolačića na serveru. Prvi argument je ime kolačića, drugi argument je vrijednost koju spremamo u kolačić i treći argument je objekt sa atributima od samog kolačića.

```
40    res.cookie("jwt", refreshToken, {
41      httpOnly: true,
42      maxAge: 24 * 60 * 60 * 1000,
43      sameSite: "none",
44      secure: true,
45    });■      You, 3 months ago • init
```

Slika 4.5. Kreiranje kolačića

Nakon slanja kolačića, ako inspektamo web stranicu i otvorimo *Storage* → *Cookies* može se vidjeti kreirani kolačić sa prosljeđenim argumentima što se vidi na slici 4.6.



Slika 4.6. Kolačić u web pregledniku

Prilikom izrade Express servera obavezno moramo uključit "cookie-parser" *middleware* zato što express ne zna što su kolačići i ne može ih prepoznati, stoga korištenjem *cookie-parser*-a na poslani

request sa frontenda dodan je kolačić te samim time je omogućeno Expressu da pristupi kolačićima što se vidi na slici 4.7.

```
14
15  // Importing cookie parser
16  const cookieParser = require("cookie- parser");
17
18  // Cookies handler
19  app.use(cookieParser());      You, 3 months ago • init
20
```

Slika 4.7. Middleware za kolačiće

4.3. CORS

CORS je mehanizam koji omogućava serveru da definira koje domene, osim njega samoga, mogu pristupiti njegovim *endpoint*-ima. CORS je nužan za sigurnu komunikaciju dviju aplikacija koje koriste različite domene, u napravljenoj aplikaciji to su klijent i server. Konfiguriranjem određenih svojstava u *response headeru* vrlo lako je postaviti CORS ograničenja. Na slici 4.8. se vidi konfiguracija CORS *middleware* na serveru.

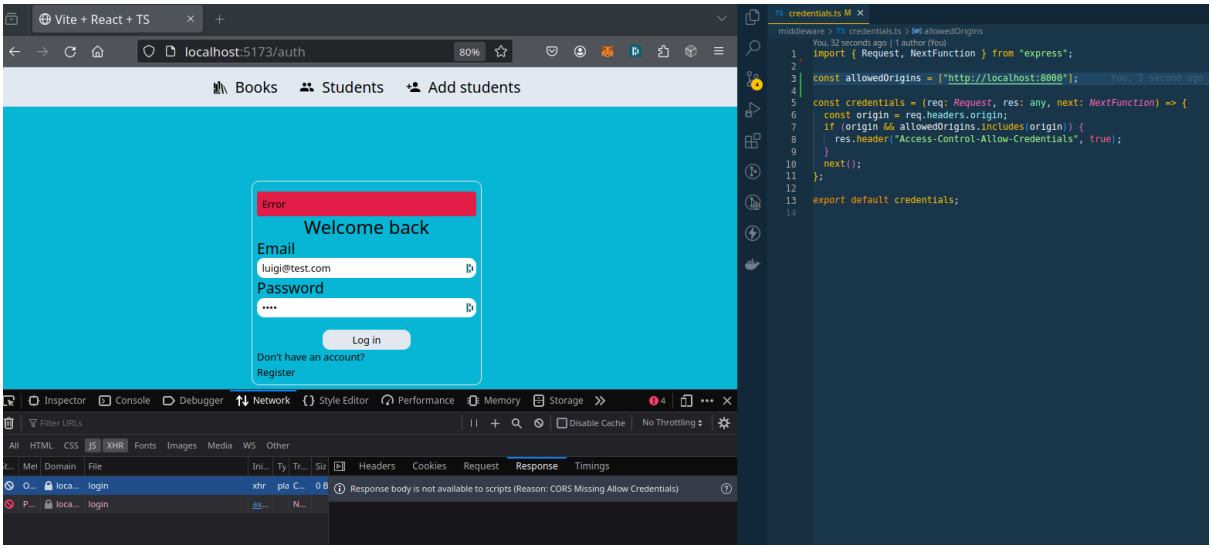
```
11
12  app.use(credentials);
13
14  app.use(cors(corsOptions));      You, 3 months ago • init
15
```

Slika 4.8. CORS middleware

Middleware "credentials" na slici 4.9. provjerava dali domena koja želi pristupiti podatcima ima dozvolu, te ako ima dozvolu postavlja "Access-Control-Allow-Credentials" u istinito što govori web pregledniku da server prihvaća *credentials* sa druge domene. *Credentials* mogu biti kolačići ili *authentication header*.

Slika 4.9. Credentials middleware

Forsiranjem erora, odnosno promjenom porta naše lokalne domene iz "allowedOrigins" liste. Dobit ćemo CORS eror "CORS Missing Allow Credentials", vidljiv na slici 4.10.



Slika 4.10. CORS eror

Postavljem CORS *middleware*-a i prosljeđivanjem "corsOptions" objetka moguće je manipulirati CORS svojstvima: *origin*, *methods*, *optionsSuccessStatus* te mnogim drugima, što se vidi na slici 4.11. ispod.

Slika 4.11. Postavljanje CORS svojstva

4.4. Status kodovi

Http status kodovi su predefinirani kodovi uz pomoć kojih obaviještavamo programera ili korisnika o statusu *request*-a odnosno njegovoj uspješnosti. Podijeljeni su u 5 kategorija, opisat će se 3 kategorije koje se najviše koriste, a to su: uspješan(200-299), greška na klijentu(400-499) i greška na serveru(500-599).

Kodovi koji prikazuju uspješnost *requesta* su oni između 200 i 299. Neki od korištenih u aplikaciji su:

- 200 (“Ok”) - *request* je uspješan, u slučaju *GET* metode znači da su se dobili željeni podatci
- 201 (“Created”) - najčešće se koristi nakon uspješno kreiranih podataka u bazi prilikom *POST* ili *PUT* metode
- 204 (“NO CONTENT”) - *request* je uspješan, ali nema odgovora, može se koristiti prilikom *DELETE* metode.

Status kodovi koji nas obavještavaju o grešci bila to klijentska ili serverksa su 400-499 i 500-599. Razlika je u tome što kodovi 400 nalažu da je izbila greška na strani klijenta, dok status kodovi 500 opisuju grešku na serveru. Najlakše je razumijeti razliku ove dvije vrste greške na primjeru interakcije korisnika sa aplikacijom.

Korisnik se želi ulogirati u sustav, ali prilikom logina dobije status kod 401 “unauthorized”. Iako se 401 zove *unauthorized* u praski on znači *unauthenticated* što nam govori da korisničko ime ili šifra nisu ispravni, stoga korisnik nije autentificiran. Na slici 4.12. je prikazan kontroler zadužen za *login*, trenutno će se zanemariti programski kod koji se odnosi na generiranje tokena. Nakon što se uspješno na server dobilo email adresu i lozinku. Probat će se naći korisnika sa tom email adresom, ako taj korisnik ne postoji, vratit ćemo 401. Ako pak korisnik postoji usporediti ćemo hashiranu lozinku sa lozinkom koja je spremljena u bazi, ako su jednake login je uspješan, a ako nisu opet ćemo vratiti 401 zato što je korisnik poslao pogrešnu lozinku.

```
const handleLogin = async (
  { body: { email, pwd } }: Request,
  res: Response
) => {
  // Find user
  const user = await adminService.findUserBy(UserFindersKey.EMAIL, email);
  // If there is no user in db it means that is unauthenicated
  if (!user) return res.sendStatus(401);
  // Check match
  const match = await bcrypt.compare(pwd, user.pwd);
  if (match) {
    // Find roles
    const role = user.role;
    // Create accessToken and refresh token
    const accessToken = generateJWT({···
    });
    const refreshToken = generateJWT({···
    });
    // Update user
    await adminService.findAndUpdateRefreshToken(email, refreshToken);

    res.cookie("jwt", refreshToken, {···
    });
    res.json({ accessToken });
  } else {
    // If passwords don't match, it is unauthenicated request    You, 1 second ago • Uncommitted changes
    res.sendStatus(401);
  }
};
```

Slika 4.12. Login kontroler

Drugi slučaj: Korisnik ponavlja identičan proces, ali dobiva status kode 500 server greška. 500 govori da je došlo do greške na serveru, ali server ne može točno odrediti što se desilo. Neki od mogućih situaciju su: problemi sa *gateway*-om, prekid konekcije servera i baze itd.

Uz objašnjeni status kod 401 spomenit će se još dva status koda iz grupe 400, a to su 403 *forbidden* i 404 *not found*.

403 status kod ili *forbidden* nam govori da korisnik nema prava pristupa na ciljani *endpoint* drugim riječima nije autoriziran. Razlika 403 i već spomenutog 401 je u tome što 403 govori da prepoznaje korisnika, ali taj korisnik nema prava pristupa. I posljednji, najpoznatiji 404 *not found*, nam govori da ruta kojoj korisnik želi pristupiti ne postoji.

5. AUTENTIFIKACIJA I AUTORIZACIJA

U ovom poglavlju će se detaljno opisati kako u aplikaciji radi autentifikacija i autorizacija. Glavne stavke autentifikacije na koje se mora paziti su: spremanje korisnikove lozinke, te na koji način osigurati da svaki sljedeći *request* od strane logiranog korisnika bude autentificiran na način da korisnik ne mora svaki puta kad želi pristupiti zaštićenom *endpoint*-u ponovno unositi email i lozinku. Prvi problem se rješava vrlo jednostavno, a to je da u bazu spremamo haširane lozinke, tako da ni vlasinici baze podataka ne znaju izvornu lozinku. Već se prilikom *logina* uspoređuje haš iz baze sa poslanom lozinkom. Drugi problem se ne može tako jednostavno riješiti, solucija je da nakon uspješne prijave vratimo korisnikove osjetljive podatke, i dobivene podatke šaljemo prilikom svakog sljedećeg *request*-a i na taj način vršimo provjeru, ali tu se nailazi na veliki problem, a to je sigurnost. Sve što je vraćeno na klijentsku stranu odnosno na frontend je dostupno napadaču. Problem se rješava pomoću JWT. Pomoću JWT ćemo osigurati da korisnik sa sigurnošću može pristupati zaštićenim endpointima na serveru, a da to ne utječe na njegovo korištenje aplikacije. Autorizacija ili provjera da li korisnik ima prava pristupa određenom endpointu će se također riješiti preko JWT koji će sadržavati rolu korisnika.

5.1. JWT

JWT [3] je prihvaćeni standard za siguran prijenos informacija između dvije strane u formatu JSON objekta. JWT garantira sigurnost zato što je digitalno potpisan sa HMAC, RSA algoritmom. JWT se sastoji od tri dijela: zaglavlja, podataka i digitalnog potpisa. Zaglavlje se sastoji od dva dijela, a to su: vrsta tokena i algoritma korištenog za digitalni potpis. Podatci su informacije koje spremamo u token. U izrađenom Express serveru će se koristiti "jsonwebtoken" paket koji će nam pomoći prilikom potpisavinja i verificiranja tokena u Nodejs. Na slici 5.1. je vidljivo potpisavanje tokena korištenjem instaliranog paketa. *Sign* funkcija prima tri parametra, prvi argument je objekt sa podatcima koje želimo potpisat, drugi je tajni ključ što je *base64* kodni string koji nam služi pri potpisivanju tokena, tajni ključ mora ostati tajna i mora biti spremljen kao env. varijabla uprotivnom će svatko moći dekodirati naš token i posljednji argument je vrijeme trajanja tokena o ćemo će se reći malo detaljnije kasnije u radu.

```
const jwt = require("jsonwebtoken");

const user = {
  email: "dumbmail@mail.com",
  role: "admin",
};

const ACCESS_TOKEN_DURATION = "60s";

// Access token creation
const accessToken = jwt.sign({
  // Payload
  data: {
    UserInfo: {
      email: user.email,
      role: user.role,
    },
  },
  // ENV token secret
  tokenSecret: process.env.ACCESS_TOKEN_SECRET,
  // Token duration
  tokenDuration: ACCESS_TOKEN_DURATION,
});
```

Slika 5.1. Kreirenje JWT

Nakon prikazanog kreiranja tokena sljedeći korak je verificiranje tokena, to će se raditi pomoću *verify* funkcije koja ima tri parametra. Prvi prosljeđeni argument je token koji želimo verificirati, drugi argument je "tokenSecret" sa kojim smo potpisali token, a treći je *callback* funkcija(u JavaScript *callback* funkcija je funkcija koja samu sebe poziva). Na slici 5.2. se vidi pojednostavljen način verificiranja tokena.

Slika 5.2. Verificiranje tokena

U produkciji, verifikacija tokena će se odvojiti u zasebni *middleware* koji će manipulirati *request-om* i *response-om*. Na opisanom primjeru je vidljiva osnovna logika korištenja tokena, nakon uspješnog logina kreira se token sa korisnikovim podatcima koji se digitalno potpisuje sa tajnim ključem, te se taj token zakači na svaki sljedeći *request* gdje se prije obrade *request*-a token dekodira i provjerava njegova validnost.

5.2. Autentifikacija

U samom uvodu u trenutno poglavlje je opisana autentifikacija općenito i na pojednostavljen način je objašnjeno kako je to izvedeno u napravljanoj aplikaciji. U ovom podpoglavlju će se ući detaljnije u proces autentifikacije i pojasnit će se kako JWT riješava taj problem. Opisati će se cijeli proces koji se dešava u pozadini nakon što korisnik unese korisničko ime i lozinku.

Autentifikacija korisnika može imati dva ishoda

- Korisnik nije autentificiran
- Korisnik je uspješno autentificiran

Na slici 5.3. se vidi prvi slučaj, nakon što se dobije *body* objekt te se iz njega destrukturira email i lozinka potrebno je provijeriti dali postoji korisnik sa poslanim email-om, te ako ne postoji vratiti 401 status kod, ako postoji provijeriti podudaranje poslane lozinke sa lozinkom spremljenoj u bazi i ako se podudaraju login je uspješan, a ako su lozinke različite vraćamo 401.

```
const handleLogin = async (
  { body: { email, pwd } }: Request,
  res: Response
) => {
  // Find user
  const user = await adminService.findUserBy(UserFindersKey.EMAIL, email);
  // If there is no user in db it means that is unauthenicated
  if (!user) return res.sendStatus(401);
  // Check match
  const match = await bcrypt.compare(pwd, user.pwd);

  if (match) {
    // DO SOME STUFF WITH USER
  } else {
    // If passwords don't match, it is unauthenicated request
    res.sendStatus(401);
  }      You, 1 second ago • Uncommitted changes
```

Slika 5.3. Neuspješna autentifikacija

Nakon uspješne provjere i autentifikacije korisnika ulazimo u drugi kompleksniji korak. U drugom koraku će se kreirati dva tokena: pristupni i refresh. Naime dva spomenuta tokena će imati različitu funkciju i različito trajanje. Pristupni će sadržavati korisnikov email i rolu, njegovo trajanje će biti 15 minuta. Pristupni token će se slati na frontend nakon uspješnog logina gdje će se on spremiti u memoriju. Te će se token pristupa zakačiti na svaki novi *request* koji korisnik napravi i provjerit će se njegova validnost na serveru. Također prilikom logina nakon kreiranog pristupnog tokena kreirat će se i refresh token. Refresh će sadržavati email od korisnika i njegovo će trajajanje biti 1 dan, on će se za razliku od pristupnog tokena spremiti u bazu i postaviti će kolačić sa njegovim sadržajem. Kolačić će imati postavljene *httpOnly* i *secure* svojstva u *true* tako da je siguran i nedostupan *frontend*-u. Refresh token ne smije biti dosupan frontendu jer bi stim omogućili da napadač ima pristup zaštićenim podatcima cijeli dan. Kreiranje oba tokena te postavljenje refresh tokena u kolačić te vraćanje pristupnog tokena u *respons*-u objektu se vidi na slici 5.4.

```
if (match) {
  // Find roles
  const role = user.role;
  // Access token creation
  const accessToken = generateJWT({
    // Payload
    data: {
      UserInfo: {
        email: user.email,
        role: role,
      },
    },
    // ENV token secret
    tokenSecret: process.env.ACCESS_TOKEN_SECRET,
    // Token duration
    tokenDuration: ACCESS_TOKEN_DURATION,
  });
  const refreshToken = generateJWT({
    // Payload
    data: { email: user.email },
    // Token secret
    tokenSecret: process.env.REFRESH_TOKEN_SECRET,
    // Token duration
    tokenDuration: REFRESH_TOKEN_DURATION,
  });
  // Update user
  await adminService.findAndUpdateRefreshToken(email, refreshToken);

  // Set refresh token to httpOnly cookie
  res.cookie("jwt", refreshToken, {
    httpOnly: true,
    maxAge: 24 * 60 * 60 * 1000,
    sameSite: "none",
    secure: true,
  });
  // Return accessToken to frontend      You, 1 second ago • Uncommitted changes
  res.json({ accessToken });
```

Slika 5.4. Uspješna autentifikacija

Token pristupa koji ima kratko trajanje će se koristiti za autentifikaciju korisnika, u slučaju isteka tokena pristupa izvršit će se *request* na refresh *endpoint* čija je svrha generiranje novog pristupnog tokena uz provjeru validnosti refresh tokena. Zadaća refresh *endpoint*-a je provjeriti postojanost kolačića u kojem je spremljen refresh token, te će se pronaći korisnik sa pripadajučim poslanim refresh tokenom, nakon toga će se izvršiti provjera validnosti tokena te ako je token validan generirati će se novi pristupni token sa korisnikovim podatcima te će se novokreirani pristupni token poslati nazad na *frontend*, što se vidi na slici 5.5.

Slika 5.5. Refresh kontroler

Implementacija prikazane *backend* logike na *frontend* dijelu aplikacije će se napraviti na sljedeći način. Nakon uspješne autentifikacije, na *frontend* će se vratiti pristupni token, te će se njega spremiti u *state*. *State* možemo zamisliti kao trenutnu memoriju, stoga doklegod je *frontend* otvoren u web pregledniku token će postojati, ali nakon zatvaranja ili osvježavanja preglednika sve iz memorije će nestati pa tako i token. Nakon spremanja pristupnog tokena u memoriju taj isti token će se zakačiti na svaki sljedeći *request*. Pristupni token se će spremiti u takozvani *Bearer*

token. *Bearer token* je najkorištenija metoda autentifikacije i autorizacije preko https protokola. Svaki *request* sadrži zaglavlje koji služi za postavljanje dodatnih informacija vezanih za *request*. *Bearer token* se nalazi u zaglavlju pod *authorization* svojstvom. Na slici 5.6. se vidi *frontend* logika postavljanje tokena u *headers.authorization* odnosno *Bearer token*.

```
// Get access token from react state
const { auth } = useAuthContext();
// Add access token to headers.authorization
const requestInt = axiosPrivate.interceptors.request.use(
  (config) => {
    // First request
    if (!config.headers['Authorization']) {
      config.headers['Authorization'] = `Bearer ${auth.accessToken}`;
    }
    return config;
  },
  (error) => Promise.reject(error)
);
```

Slika 5.6. Postavljanje Bearer tokena na frontendu

Nakon postavljanja tokena u *autorization headers* sljedeće će se pokazati verifikacija tog tokena na *backend*-u. Na *backend*-u će se kreirati *middleware* "verifyAccessJWT" koji će iz *request*-a koji dolazi sa *frontenda* odvojiti token pristupa koji je spremljen u *headers.authorization*, te će tom istom tokenu provijeriti validnost, što je vidljivo na slici 5.7.

Slika 5.7. Verifikacija pristupnog tokena na backendu

Verifikacija tokena je potrebna samo na zaštićenim *endpoint*-ima, nema smisla provijeravati validnost na login *endpoint*-u kad korisnik još uopće nema pristupni token. Express aplikacija funkcionira na način da se programski kod izvršava od vrha prema dnu. Tako da "verifyAccessJWT" *middleware* je potrebno postaviti iznad ruta koje želimo zaštiti, a to su "/api/students" i "/api/books" zato što te rute nisu dostupne ne autentificiranim korisnicima. Na slici 5.8. se vide sve rute te se također jasno vidi da zaštićene rute iznad sebe imaju "verifyAccessJWT" *middleware*.

```
// CREATING EXPRESS APP
const app = express();

app.use(credentials);

app.use(cors(corsOptions));

// Express parser
app.use(express.json());

// Cookies handler
app.use(cookieParser());

// LOGIN AND REGISTRATION ROUTES
app.use("/register", require("./routes/register"));
app.use("/login", require("./routes/login"));
app.use("/logout", require("./routes/logout"));
app.use("/refresh", require("./routes/refresh"));

// ACCESS TOKEN VERIFICATION
app.use(verifyAccessJWT);
// API ROUTES
app.use("/api/students", require("./routes/api/students"));
app.use("/api/books", require("./routes/api/books"));

// SETTING PORT      You, 1 second ago • Uncommitted changes
app.listen(3000, () =>
  console.log("REST API server running at: http://localhost:3000")
);
```

Slika 5.8. Provjera autentifikacije prije API ruta

Na slici 5.8. je također vidljiv "/refresh" *endpoint*, njegova uloga je kreiranje novog pristupnog tokena u slučaju da poslani nije valjan, stoga na *frontend*-u će se napraviti logika koja će pozivati refresh *endpoint*. Već je pokazano na koji se način dodaje pristupni token u *request header* na *frontend*-u, sada će se u istoj funkciji napraviti i obrada erora. Obrada erora će provijeriti dobiveni odgovor sa *backend*-a te će na osnovu erora napraviti novi *request* na *backend*, točnije na "/refresh" endpoint koji će generirati novi pristupni token koji će zamijeniti ne valjani token. Nakon izvršene logike napravit će se novi *request* sa novim valjanim tokenom. Na slici 5.9. se vidi cijela funkcija na *frontendu* koja obavlja opisanu logiku.

useAddAccToken.tsx M

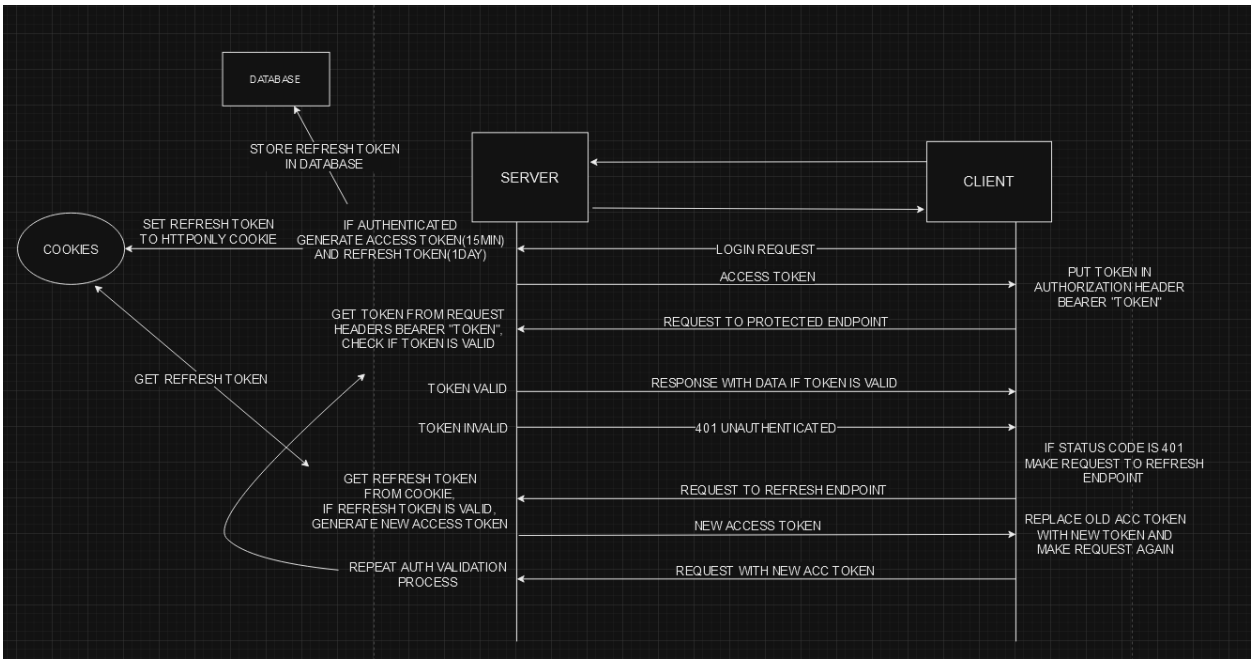
e-learning-app > src > hooks > useAddAccToken.tsx > useAddAccToken > useEffect() callback > response

You, 1 second ago | 1 author (You)

```tsx
import { useEffect } from 'react';
import { useAuthContext } from '../context/auth/context';
import useRefresh from '../hooks/useRefresh';
import { AxiosInstance } from 'axios';

const useAddAccToken = (axiosPrivate: AxiosInstance) => {
  const { auth } = useAuthContext();
  const handleRefresh = useRefresh();

  useEffect(() => {
    // Do add accesstoken to headers
    const requestInt = axiosPrivate.interceptors.request.use(
      (config) => {
        // First request
        if (!config.headers['Authorization']) {
          config.headers['Authorization'] = `Bearer ${auth.accessToken}`;
        }
        return config;
      },
      (error) => Promise.reject(error)
    );
    // Response interceptors
    const responseInt = axiosPrivate.interceptors.response.use(
      (response) => response,
      // Interceptor error
      async (error) => {
        const prevRequest = error?.config;
        if (error.response.status === 401 && !prevRequest._retry) {
          prevRequest._retry = true;
          // Make request to "/refresh" endpoint      You, 1 second ago • Uncom
          const newAccToken = await handleRefresh();
          prevRequest.headers['Authorization'] = `Bearer ${newAccToken}`;
          return axiosPrivate(prevRequest);
        }
        return Promise.reject(error);
      }
    );
    //  Remove previous intercepotrs
    return () => {
      axiosPrivate.interceptors.request.eject(requestInt);
      axiosPrivate.interceptors.response.eject(responseInt);
    };
  }, [auth, handleRefresh]);
  return axiosPrivate;
};

export default useAddAccToken;
```

Slika 5.9. Implementacija autentifikacije na frontendu

Na slici 5.10. je grafički prikazan prethodno opisani tijek autentifikacije.



Slika 5.10. Grafički prikaz autentifikacije

U ovom poglavlju se ušlo detaljnije što se sve dešava u pozadini prilikom korištenja aplikacije, a vezano je za autentifikaciju, samim time je objašnjen najkompleksniji dio aplikacije. U nastavku će se objasniti logika autorizacije, koja će koristiti slične logičke principe, ali na malo jednostavniji način.

5.3. Autorizacija

Autorizacija ili pravo pristupa je način na koji korisniku omogućujemo ili zabranjujemo da pristupi određenom *endpoint*-u u našoj aplikaciji. U aplikaciji postoje 3 role, a to su: “admin”, “student” i “teacher”. Osim ruta vezanih za autentifikaciju kreirana su još dvije api rute: “/students” i “/books”. Pristupanje *endpoint*-ima vezanih za autentifikaciju (login, register, logout) nema smisla postavljati prava pristupa zato što korisnik još nije ulogiran u sustav. Dok u slučaju API *endpoint*-a na kojima će se vršiti CRUD, autorizacija je obavezna. Admin ili rola sa najvećim pravima pristupa ima mogućnost vršiti sve CRUD operacije nad API *endpoint*-ima. Dok slučaj kod druge dvije role je malo drugačiji. Rola “teacher” će imati viša prava pristupa od “student”, ali niža od “admina”. Dok najniža rola “student”, koja se dodjeljuje automatski svakom novom registriranom korisniku će moći pristupiti samo određenim operacijama nad “/books” *endpoint*-u. Provjera role

će se također vršiti kroz *middleware* funkciju, nije praktično pozvati *middleware* za autorizaciju prije cijelog *endpoint*-a zato što trebamo biti malo precizniji prilikom autorizacije, zato će se to učiniti unutar rute odnosno zasebno nad svakim *endpoint*-om. Prilikom registracije, odnosno nakon što se kreira novi korisnik u bazi njemu se dodijeli rola "student". Sada se nailazi na isti problem kao i kod autentifikacije, na koji način slati rolu korisnika prilikom svakog sljedećeg *request*-¸a na siguran način, riješenje je također JWT pristupni token u kojeg će se osim email, također postaviti rola korisnika nakon njegove autentifikacije što se vidi na slici 5.11. u login kontroleru.

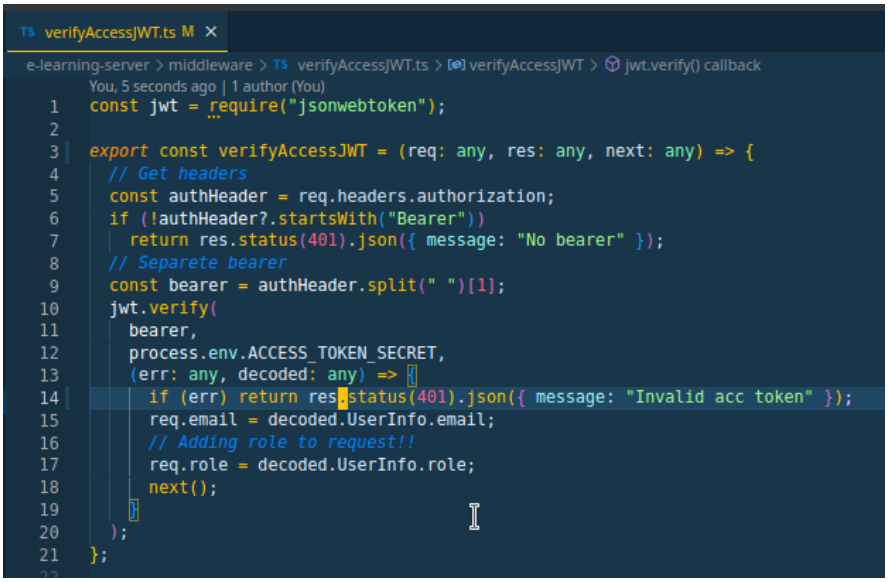
```
const handleLogin = async (
  { body: { email, pwd } }: Request,
  res: Response
) => {
  // Find user
  const user = await userService.findUserBy(UserFindersKey.EMAIL, email);
  // Unauthenticate
  if (!user) return res.sendStatus(401);
  // Check match
  const match = await bcrypt.compare(pwd, user.pwd);
  if (match) {
    // Find roles
    const role = user.role;
    // Create accessToken with user role and email as payload
    const accessToken = generateJWT({
      data: {
        UserInfo: {
          email: user.email,
          role: role,
        },
      },
      tokenSecret: process.env.ACCESS_TOKEN_SECRET,
      tokenDuration: ACCESS_TOKEN_DURATION,
    });      You, 5 months ago • init
```

Slika 5.11. Spremanje korisnikove role u JWT

Nakon kreiranja pristupnog tokena, potrebno je kreirati *middleware* koji će provijeriti rolu. *Middleware* će primiti listu sa rolama kojima je omogućen pristup te će iz poslanog *request*-a izvući rolu i provijeriti da li je ona u prosljeđenoj listi, ako se rola ne nalazi u listi vratit će se 403 što znači da korisnik nema prava pristupa, što je vidljivo na slici 5.12.

Slika 5.12. Middleware za provjeru role

Na prethodnoj slici na liniji 5 programskog koda vidi se da je rola dobivena iz “req.role” svojstva, po defaultu *request* odnosno req u prikazanom slučaju nema “role” svojstvo. Zato je obavezno potrebno u *middleware*-u za provjeru autentifikacije postaviti *req.role* sa vrijednosti koju smo dobili iz tokena nakon dekodiranja što je vidljivo na slici 5.13. u “verifyAccessJWT” middleware. Zbog spomenutog mehanizma je vrlo bitno da se *middleware* “verifyAccessJWT” poziva prije “verifyRoles” *middleware*, uprotivnom “verifyRoles” *middleware* neće imati *req.role* svojstvo koje koristi za provijeru role.

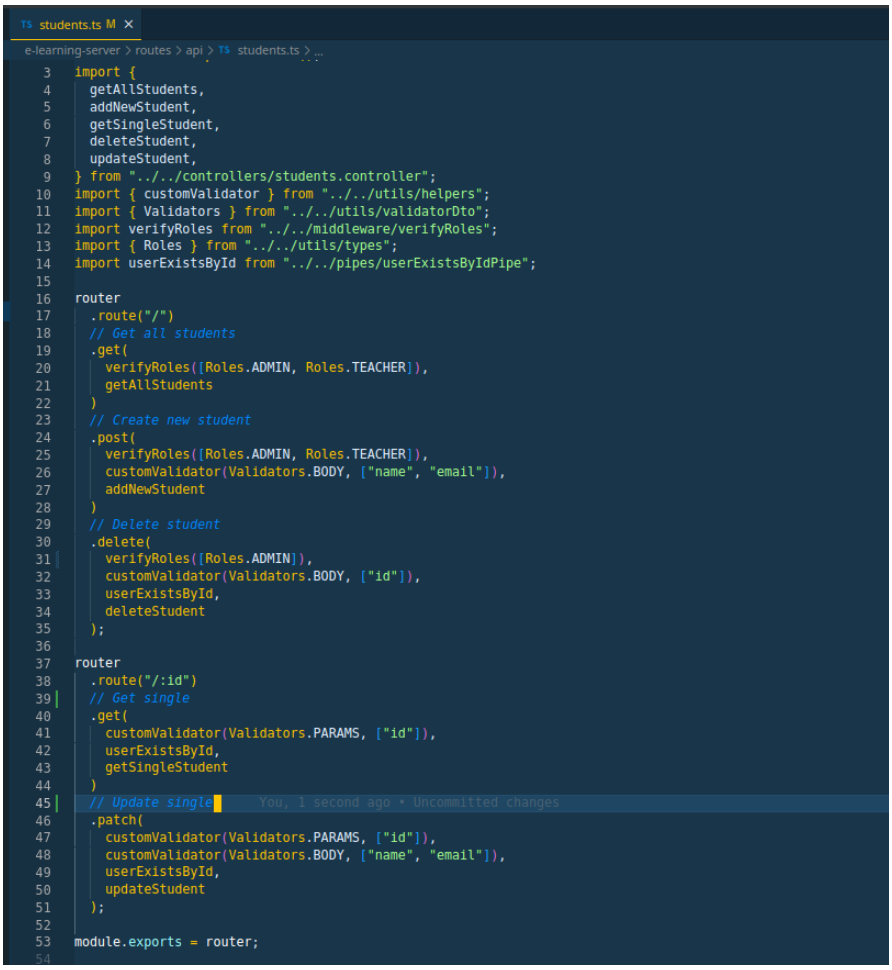


Slika 5.13. Dodavanje role u request objekt

Nakon primjene objašnjene logike moguće je pozivati “verifyRoles” *middleware* nad svakom operacijom u određenom *endpoint*-u.

5.3.1. Implementacija autorizacije

U ovom poglavlju će se prikazati implementacija autorizacije nad API-ima. Prvi API koji će se uzeti za primjer je “students”. Students API se sastoji od 2 *endpoint*-a, koji se vide na slici 5.14.



Slika 5.14. Students endpointi

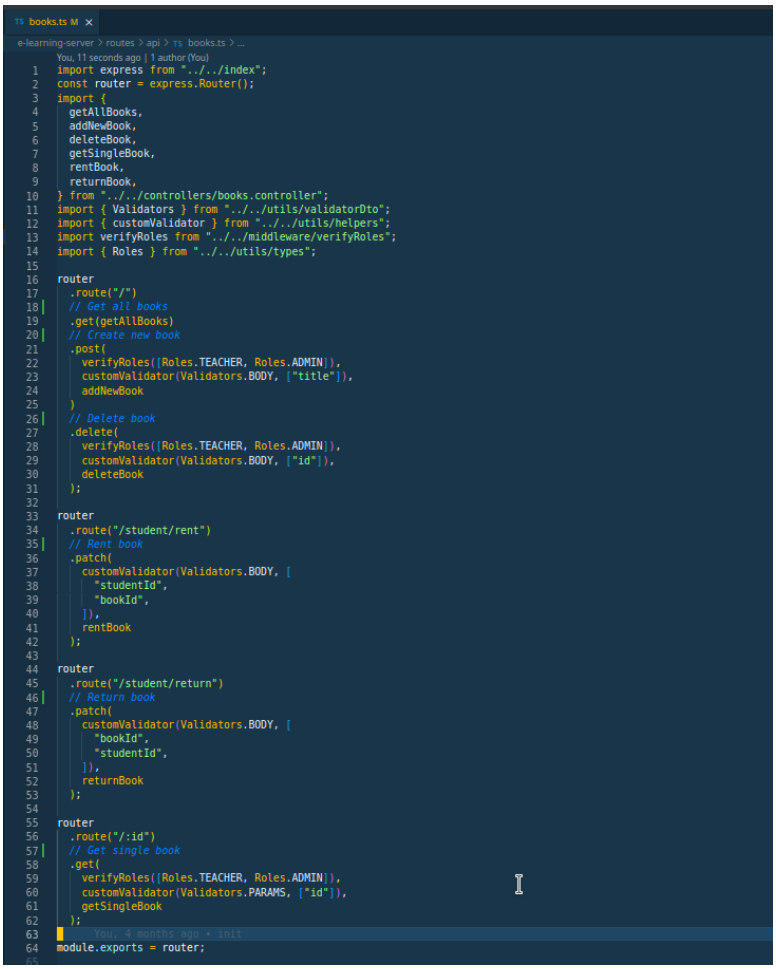
Svaki *endpoint* se sastoji od *path*-a, metode, *middleware* i na kraju kontrolera koji obrađuje *request* i vraća odgovor. Na nekim endpointima je korišten "customValidator" *middleware* koji nam služi za validaciju podataka, konkretno sa slike to su: *body* i *params*. Nadalje će se opisati autorizacija nad prikazanim *endpoint*-a. Prvi endpoint sadrži tri metode, dok drugi endpoint sadrži dvije metode.

- GET "/api/students" (getAllStudents), endpoint za dohvaćanje svih studenata, ovaj *endpoint* ima ograničen pristup tako da samo korisnici sa rolama "admin" i "teacher" mogu pristupiti ovom *endpoint*-u.

- POST “/api/students” (addNewStudent), *endpoint* za kreiranje novog studenta, također ima ograničen pristup gdje samo “admin” i “teacher” role imaju pristup.
- DELETE “/api/students” (deleteStudent), endpoint za brisanje studenta, samo “admin” ima pravo na brisanje studenta.
- GET “api/students/:id” (getSingleStudent), endpoint za dohvaćanje jednog studenta, jedino ograničenje ovog *endpoint* je da korisnik ako ima rolu “student” može dohvatiti jedino samog sebe, to znači da korisnik sa rolom “student” ne može dohvatiti druge studente.
- PATCH “api/students/:id” (updateStudent) *endpoint* za promjenu podataka korisnika, jednaka ograničenja kao i kod prethodnog *endpoint*-a.

Logika koja stoji iza spomenutih restrikcija je vrlo jednostavna, “admin” kao korisnik sa najvišim pravima pristupa može pristupiti svim *endpoint*-ima. Korisnik “teacher” može pristupiti gotovo svim *endpoint*-ima kao i “admin”, osm što ne može brisati studenta zato što je to radnja koja zahtjeva pristup “admina”. Te korisnik “student” sa najnižim pravima pristupa može samo pristupiti isključivo svojim podatcima.

Drugi API koji će se objasniti je API “books”. Books sadrži 4 *endpoint*-a, što je vidljivo na slici 5.15.



Slika 5.15. Books endpointi

- GET “/api/books/” (getAllBooks), nema restrikcija što se tiče autorizacije,svi korisnici mogu dohvatiti sve dostupne knjige.
- POST “/api/books/” (addNewBook), sve role osim role “student” mogu kreirati novu knjigu u bazi podataka.
- DELETE “/api/books/” (deleteBook), isto kao i u prethodnom *endpoint*-u, svi osim “student” mogu brisati knjige.
- PATCH “/api/books/student/rent” (rentBook), nema restrikcija, svi korisnici mogu pozvati ovaj *endpoint*.
- PATCH “/api/books/student/return” (returnBook), isto kao i u prethodnom *endpoint*-u.

- GET “/api/books/:id” (getSingleBook), svi osim “student” mogu pristupiti ovom *endpoint*-u. Student može pristupiti ovom *endpoint*-u samo ako želi dohvatiti knjigu koja nije uzeta.

Logika je slična kao i kod “students” API, student nema pravo brisati i dodavati nove knjige, te samo može pristupiti *endpoint*-ima sa GET metodom koja dohvaća određene podatke, dok također student može vršiti akcije za posuđivanje/vraćanje knjige. Ostalim *endpoint*-ima mogu pristupiti korisnici sa rolama “admin” i “teacher”.

6. ZAKLJUČAK

Kroz ovaj rad su detaljno opisani i primjenjeni glavne stavke govoto svake web aplikacije, a to su autentifikacija i autorizacija, objašnjeni principi i korištene logike se mogu primjeniti na gotovo svaku aplikaciju koja zahtjeva registraciju korisnika tako da ova aplikacija je odličan primjer kako se to primjenilo na konkretnom slučaju sustava za e-učenje. Razvojem tehnologije raste i razina sigurnosti stoga je na konkretnom primjeru provijere identiteta korisnika implementirana visoka sigurnost bez narušavanja korištenja same aplikacije odnosno UX (engl. *User experience*). Sam proces provjere identiteta nije jednostavan, ali je zato lako primjenjiv na bilo koju aplikaciju. Izuzev sigurnosti, opisan je proces kreiranja REST API nad kojim se vrše sve CRUD operacije. Aplikaciji se može dodatno podignuti razina sigurnosti na način da se napravi takozvana rotacija refresh tokena, odnosno prilikom pozivanja refresh *endpoint*-a da se nakon kreiranja pristupnog tokena kreira i novi refresh token, kreiranjem novog refresh tokena se dodatno otežalo napadaču da pristupi zaštićenim podatcima.

LITERATURA

[1] Simpson, K. (2020) *You don't know JS Yet: Get started,* 2. izdanje.

[2] Gupta, L. (2023) *What is REST?*, https://restfulapi.net/.

[3] Auth0 by Okta *Introduction to JSON Web Tokens*, *https://jwt.io/introduction*.

POPIS SLIKA

Slika 1.1. Autentifikacija i autorizacija 4
Slika 2.1. Kreiranje servera 6
Slika 2.2. Prisma schema 7
Slika 2.3. Typescript eror 8
Slika 3.1. API rute 10
Slika 3.2. Endpoint books 10
Slika 4.1. Arhitektura servera 11
Slika 4.2. Middleware za validaciju 12
Slika 4.3. Students kontroler 13
Slika 4.4. Kreiranje studenta 13
Slika 4.5. Kreiranje kolačića 14
Slika 4.6. Kolačić u web pregledniku 14
Slika 4.7. Middleware za kolačiće 15
Slika 4.8. CORS middleware 15
Slika 4.9. Credentials middleware 16
Slika 4.10. CORS eror 16
Slika 4.11. Postavljanje CORS svojstva 17
Slika 4.12. Login kontroler 18
Slika 5.1. Kreirenje JWT 21
Slika 5.2. Verificiranje tokena 22
Slika 5.3. Neuspješna autentifikacija 23
Slika 5.4. Uspješna autentifikacija 24
Slika 5.5. Refresh kontroler 25
Slika 5.6. Postavljanje Bearer tokena na frontendu 26
Slika 5.7. Verifikacija pristupnog tokena na backendu 27
Slika 5.8. Provjera autentifikacije prije API ruta 28
Slika 5.9. Implementacija autentifikacije na frontendu 29
Slika 5.10. Grafički prikaz autentifikacije 30
Slika 5.11. Spremanje korisnikove role u JWT 31

Slika 5.12. Middleware za provjeru role 32
Slika 5.13. Dodavanje role u request objekt 33
Slika 5.14. Students endpointi 34
Slika 5.15. Books endpointi 36