Izrada backend komponenti za digitalnu platformu "Physio"

Hadžić, Adis

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

ADIS HADŽIĆ

IZRADA BACKEND KOMPONENTI ZA DIGITALNU PLATFORMU “PHYSIO”

Završni rad

Pula, 16. rujna 2021.

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

ADIS HADŽIĆ

IZRADA BACKEND KOMPONENTI ZA DIGITALNU PLATFORMU “PHYSIO”

Završni rad

JMBAG: 0303082376, redovni student

STUDIJSKI SMJER: informatika

KOLEGIJ: Web aplikacije

MENTOR: doc. dr. sc. Nikola Tanković

Pula, 16. rujna 2021.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani _____Adis Hadžić_____kandidat, za prvostupnika _____informatike_____ ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljeni način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi

.

Student

Adis Hadžić

U Puli, 16. rujna 2021. godine

Sadržaj

Uvod

Platforma Physio zamišljena je od strane Kinetic Centra kao način vođenja evidencije klijenata i njihovih medicinskih podataka, vođenja rasporeda dolaska klijenata, odnosno njihovih termina. Vođeni tom idejom, kolega Domagoj Kuveždić i ja podijelili smo se tako da je kolega radio frontend dio, a ja pripadajući backend.

Trenutno se web aplikacija sastoji od nadzorne ploče (dashboard-a) na kojem su prikazane ostale komponente (kalendar, nova rezervacija, popis klijenata, usluge, financije, naplaćivanje) te koji služi kao mjesto gdje imamo sve rute na jednom mjestu. Također na dashboard-u postoji i ugrađena komponenta kalendara koji pokazuje današnji dan i termine zakazane za taj dan. Pored toga, kolega je napravio i *sidebar* koji također možemo koristiti za rutiranje po aplikaciji. Neke od funkcionalnosti koje trenutno ima aplikacija su login (mogućnost koju zasad ima samo zdravstveni djelatnik), dodavanje novih klijenata, usluga te rezervacija (termina), kao i prikazivanje i uređivanje istih

Implementacija koda je dostupna na githubu: https://github.com/adishadzic/kinetic-centar-zavrsni-backend

Radno okruženje

Što se tiče izrade *backenda* odlučio sam za kombinaciju Node.js/Express/PostgreSQL. Cijeli kod je pisan u Visual Studio Code-u, a za pokretanje samog *node* projekta koristio sam naredbu “npm init” koja inicijalizira naš project, odnosno izrađuje datoteku *package.json* koja sadrži više detalja o tome kako se zove aplikacija, koja je glavna datoteka za pokretanje, koje skripte koristimo za pokretanje servera te koje *dependencies* (dodatne pakete) koristimo.

Figure 1 package.json

Na slici iznad vidimo sve pakete koji koristi naša aplikacija, pa da malo pojasnimo čemu nam služe.

- Express – Node.js framework za izrađivanje web i mobilnih aplikacija te APIeva ○ pomaže nam da brže napravimo REST API, lakše *handle*-amo rute i http pozive.
    - API (application programming interface) – "software intermediary that allows two applications to talk to each other"

- pg (Postgres) – koristi nam za povezivanje sa bazom
- cors (Cross-Origin Resource Sharing) – paket s kojim možemo definirati s kojih sve domena prihvaćamo http pozive
- dotenv – modul koji čita environment varijable iz .env datoteke i sprema ih u process.env. Najčešće u .env spremamo varijable poput connection string-ova, jwt tajni i drugih varijabli koje ne bi trebale biti javno dostupne zbog mogućnosti raznih napada na našu aplikaciju.
- morgan – služi za ispisivanje dodatnih detalja o pojedinom http requestu u konzoli (console.log)
- jsonwebtoken - za autorizaciju kod *logina*
- bcryptjs – kriptiranje lozinki
- joi – validacija unosa (najčešće kod formi login-a ili dodavanja novog klijenta)

Pored tih paketa, koristio sam i *nodemon* kao devDependency, što znači da je to dependency koji je potreban samo prilikom *development* faze (faze razvijanja servera). Nodemon je paket uz pomoć kojeg ne moramo ručno iznova pokretati server kada nešto promijenimo, umjesto toga, napravimo našu *custom* skriptu za pokretanje servera. Npr. "dev": "nodemon . –rs", gdje točka znači da će se pokrenuti ono što imamo definirano pod "main" u *package.json*-u, u mom slučaju index.js datoteka iz src foldera. Tada sve što trebamo da bi pokrenuli tu skriptu jest napisati "npm run dev" u terminalu i spremni smo za rad.

Kako bi mogli koristiti expressove http metode, prvo moramo napraviti instancu express aplikacije kao što se može vidjeti na slici ispod.

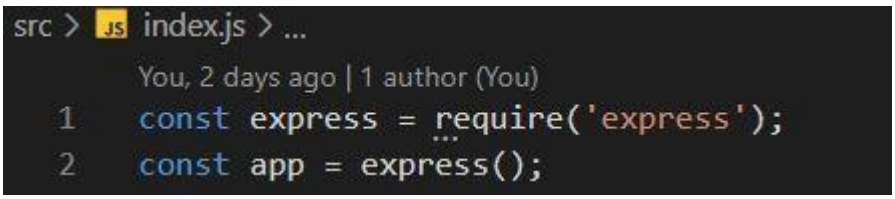

Figure 2 Instanciranje aplikacije

Zatim je potrebno odrediti port na kojem će nam backend "slušati" upite pomoću *listen()* metode.

```
13  const port = process.env.PORT;
```

Figure 4 Server port

```
40  app.listen(port, () => {
41    console.log(`🚀 Server running on port: http://localhost:${port}`);
42  });
```

Figure 3 listen()

Middleware

```
15  app.use(cors(corsConfig));
16  app.use(morgan('combined'));
17  app.use(express.json());
```

Figure 5 Middleware

Prvi middleware koji koristim je cors, koji omogućuje drugim domenama da šalju upite prema našem API-u. Na slici se može vidjeti da kao prvo imamo mogućnost definirati sve *origine* koje želimo staviti na whitelistu, te onda u funkciji samo provjeravamo sa kojeg origina dolazi request.

src > config > corsConfig.js > ...

```
You, 2 weeks ago | 1 author (You)
1   const whitelist = ['https://localhost:' + process.env.PORT, 'https://localhost:' + process.env.PORT + '/'];
2
3   const corsConfig = {
4     origin: function (origin, callback) {
5       if (whitelist.indexOf(origin) !== -1 || !origin) {
6         callback(null, true);
7       } else {
8         callback(new Error('Not allowed by CORS'));
9       }
10    },
11  };
12
13  module.exports = { corsConfig };
14
```

Figure 6 Cors whitelist

Sljedeći middleware, morgan, nam koristi tome da za svaki request u konzoli dobijemo više informacija o samom request.

```
🚀 Server running on port: http://localhost:5000
::ffff:127.0.0.1 - - [16/Sep/2021:14:21:30 +0000] "PUT /reservations/1da0ea8a-2909-4e64-b7af-515e4a82142e HTTP/1.1" 200 26 "-" "Thunder Client (https://www.thunderclient.io)"
```

Figure 7 Morgan logger middleware

I na kraju jedan od najbitnijih middleware-a/metoda što se tiče slanja body-a prema serveru, a to je express.json() middleware koji se bazira na body-parseru, što znači da one request objekte koje šaljemo u body-u prepoznaje kao JSON objekte.

Struktura backenda

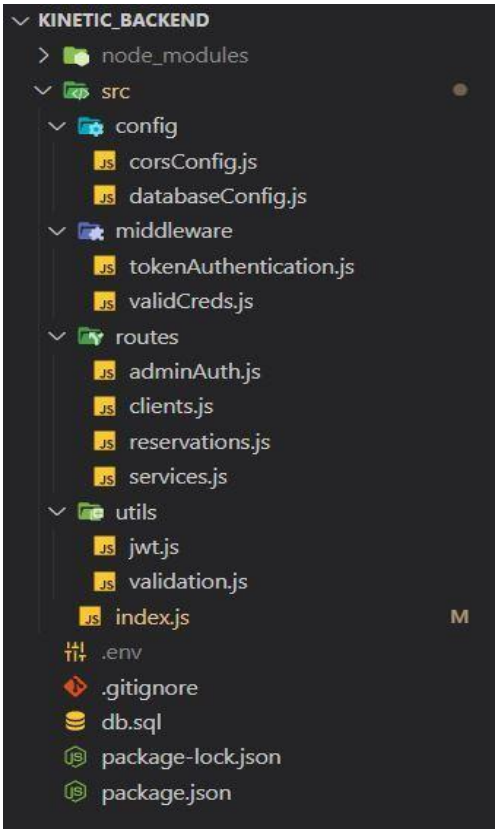

Figure 8 Folder structure

Radi lakšeg snalaženja i čitanja koda, backend sam strukturirao na sljedeći način: napravio sam jedan ulazni file – index.js u kojem instanciram aplikaciju i definiram koje sve rute koristim.

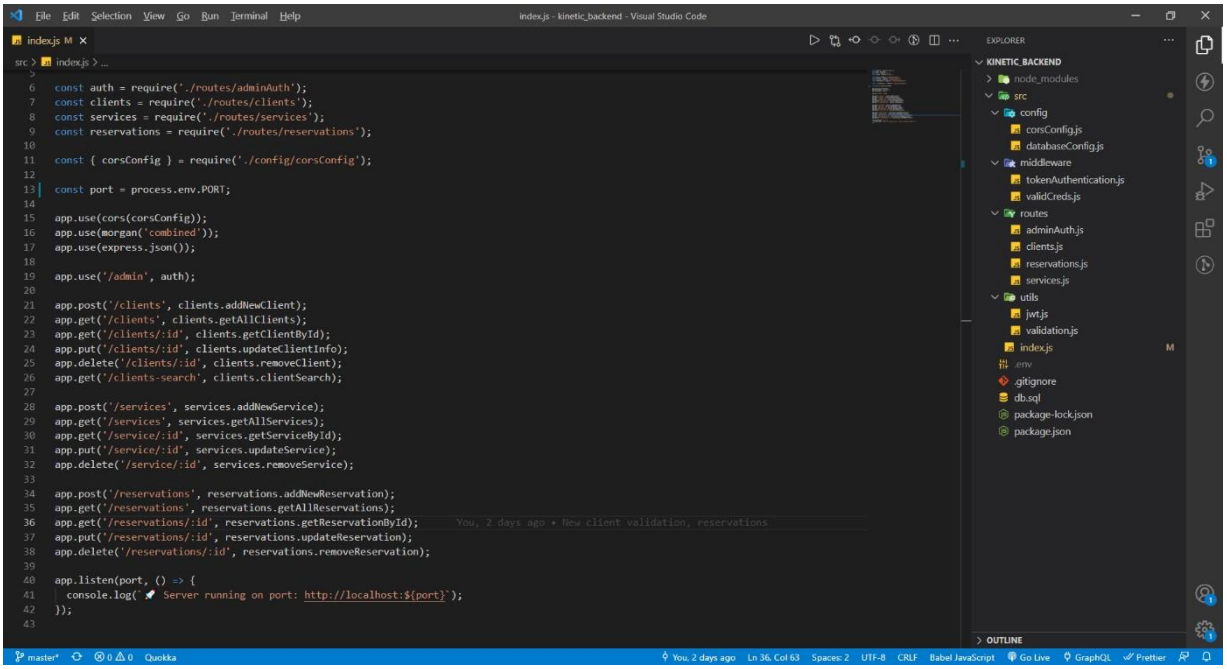

Figure 9 index.js

S time da svaka ruta ima pripadajuću datoteku u mapi *routes.*

Admin rute

Krenimo onda od adminAuth datoteke koja sadrži svu logiku koja se tiče logina korisnika i dohvaćanja svih korisnika iz baze.

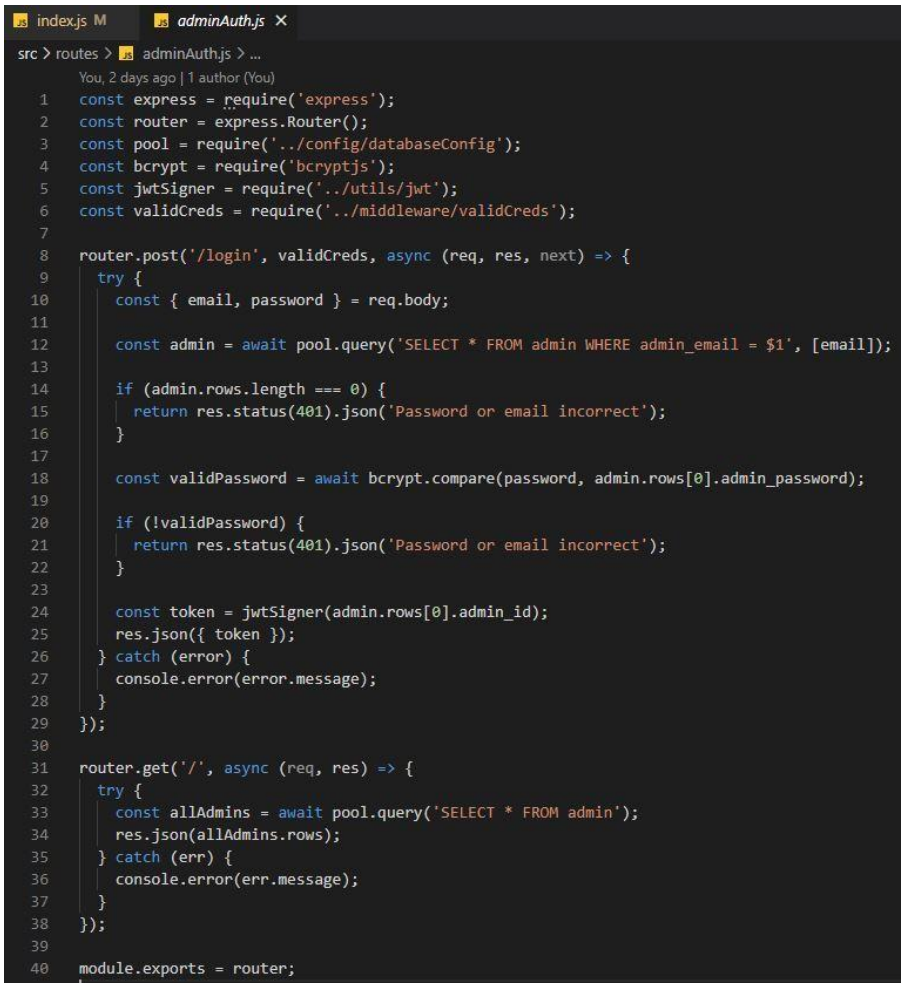

Figure 10 admin auth

Za početak, importao sam express kako bi mogao koristiti Router() metodu koja olakšava pisanje http upita. Sljedeći import se tiče konfiguracije baze podataka – pool, koji sadrži par environment varijabli (user, password, host, port, database) koje nam omogućuju spajanje na bazu i korištenje query() metode.

Pošto je process prijave u aplikaciju asinkron zbog potrebe komuniciranja sa bazom, definirao sam ovu funkciju kao asinkronu (async).

Prvi korak logina je prikupljanje ulaznih podataka od strane korisnika koji se želi ulogirati, odnosno raspakiranje request body-a. U liniji 10 (slika iznad) zapravo zatražujemo dvije stvari od korisnika – email i lozinku. Zatim u liniji 12 pomoću metode query() pišemo SQL sintaksu u kojoj tražimo da li uopće postoji taj email,

odnosno, korisnik u bazi te to spremamo u admin varijablu. Ako ne postoji, odnosno ako je admin.rows.length === 0 (nepostojeća) onda vraćamo status 401 u *response* (odgovor) i poruku “Password or email incorrect”. *Else,* odnosno ako postoji korisnik sa tim email-om u bazi, onda uspoređujemo lozinku koju smo dobili iz body-a sa lozinkom tog korisnika iz baze pomoću metode compare() od **bcrypt** modula. Treba napomenuti da je korisnik u bazi *hardkodiran*, a lozinku sam *hashirao* uz pomoć bcrypt-ovog online generatora lozinki (https://bcrypt-generator.com/). Pošto se uspoređuje dana lozinka sa lozinkom iz baze, odgovor od baze čekamo neko vrijeme, što znači da je i compare() metoda async (asinkrona).

Ako se lozinke ne podudaraju, vraćam status 401 sa porukom “Password or email incorrect”.

Razlog zašto uvijek vraćam istu poruku, za slučaj kad email nije pronađen i kad se lozinke ne podudaraju, jest daj da se potencijalnim napadačima pruži što manje detalja o erroru kako ne bi odmah znali u čemu je greška.

Na kraju, ako je email ispravan/postoji u bazi i ako se lozinke podudaraju, “potpisujemo” token korisniku uz pomoć jwtSigner funkcije koja prima ID korisnika kao parameter te ga sprema u payload object.

index.js M | jwt.js

src > utils > jwt.js > ...

You, 2 weeks ago | 1 author (You)

```
1  const jwt = require('jsonwebtoken');
2  require('dotenv').config();
3
4  function jwtSigner(admin_id) {
5    const payload = {
6      admin: admin_id,
7    };
8
9    return jwt.sign(payload, process.env.JWT_SECRET_TOKEN, { expiresIn: '12hr' });
10 }
11
12 module.exports = jwtSigner;
```

Figure 11 jwt sign function

Client rute

Figure 12 Add new client

Za kreiranje novog klijenta u bazi, koristio sam ponajprije *pool* pomoću kojeg se spajamo na bazu te clientEmailValidation funkciju koja koristi paket “Joi” koji provjerava, u slučaju emaila – da li je email stvarnog tog formata. Pored toga, sva polja su tipa *required*.

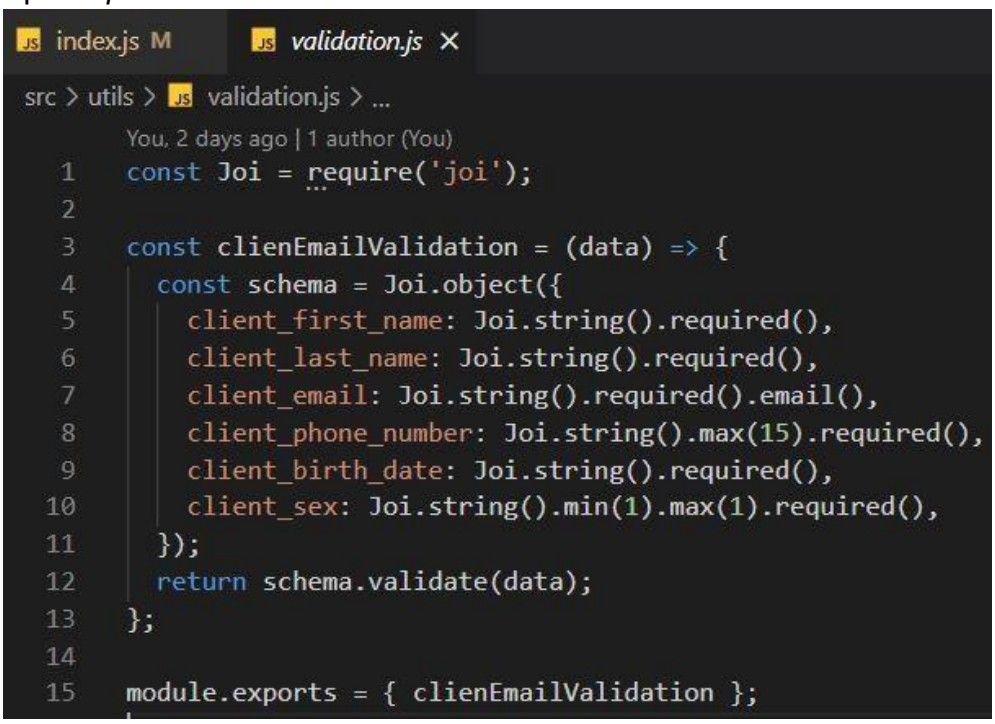

Figure 13 Joi validation

Ako nam clientEmailValidation funkcija ne baca nikakav error, onda prelazimo dalje na *trycatch* dio u kojem dohvaćamo vrijednosti iz request body-a i pomoću query() metode se izvršava SQL sintaksa za *insertanje* u tablicu. Nakon toga vraćamo

response u JSON obliku ako je sve prošlo uspješno, a ako nije, u catch dijelu ispisujemo error.

```
31  const getAllClients = async (req, res) => {
32    try {
33      const allClients = await pool.query('SELECT * FROM client');
34      res.json(allClients.rows);
35    } catch (err) {
36      console.error(err.message);
37    }
38  };
39
40  const getClientById = async (req, res) => {
41    try {
42      const { id } = req.params;
43      const client = await pool.query('SELECT * FROM client WHERE client_id = $1', [id]);
44      res.json(client.rows[0]);
45    } catch (err) {
46      console.error(err.message);
47    }
48  };
```

Figure 14 getAllClients & getClientById

Sljedeće dvije funkcije su za dohvaćanje svih klijenata i dohvaćanje pojedinog klijenta po ID-u. Te funkcije su jako slične i kod rezervacija i usluga.

Kod dohvaćanja svih klijenata zovemo SELECT * From clients query koji vraća sve elemente iz te tablice, a kod dohvaćanja po ID-u, ID dohvaćamo iz request parametara i to izgleda nešto poput /clients/:id, gdje :id označava dinamički dio te rute, odnosno ono što se smije mijenjati.

Sljedeća ruta pripada *updateanju,* tj. mijenjanju podataka pojedinog klijenta. Kao i u dohvaćanju jednog klijenta, ID klijenta dobijemo kroz parametre requesta.

U slučaju da u body-u samo šaljemo client_email, *updateati* ćemo samo email pomoću UPDATE query-a, gdje je prva vrijednost ($1) sam email, a druga ID klijenta koji smo dobili kroz parametar :id.

A ako želimo *updateati* i broj mobitela, možemo u body-u slat i email i broj mobitela.

Naposljetku, ako je sve prošlo bez greški u response šaljemo odgovor da je klijent uspješno *updatean*, a ako smo krivo unijeli key, npr client_eemail, onda u konzolu ispisujemo tu grešku.

```
50  const updateClientInfo = async (req, res) => {
51    try {
52      const { id } = req.params;
53
54      if (req.body.client_email) {
55        await pool.query('UPDATE client SET client_email = $1 WHERE client_id = $2', [
56          req.body.client_email,
57          id,
58        ]);
59      }
60
61      if (req.body.client_phone_number) {
62        await pool.query('UPDATE client SET client_phone_number = $1 WHERE client_id = $2', [
63          req.body.client_phone_number,
64          id,
65        ]);
66      }
67
68      res.json('Client`s info was updated!');
69    } catch (err) {
70      console.error(err.message);
71    }
72  };
```

Figure 15 Update client

I zadnje dvije funkcije, odnosno rute koje sam napravio su dedicirane za brisanje pojedinog klijenta i search funkcionalnost.

Što se tiče brisanja klijenta, ID ponovno zatražimo kroz parametre url-a i izvršimo DELETE query.

S druge strane, pretraga klijenata vrši se po 2 atributa, a to su client_first_name i client_last_name pomoću ILIKE operatora koji omogućuje da izvršavamo pretragu koja je case-insensitive, odnosno nije bitno da li tražimo malim ili velikim početnim slovom. Također, samo ime klijenta dobivamo iz request query-a, stringa koji se nalazi u url-u nakon znaka ?. Npr. /clients?name=John.

```
74  const removeClient = async (req, res) => {
75    try {
76      const { id } = req.params;
77      const deleteClient = await pool.query('DELETE FROM client WHERE client_id = $1', [id]);
78
79      res.json('Client was removed from the database!');
80    } catch (err) {
81      console.error(err.message);
82      res.json('Wtf');
83    }
84  };
85
86  const clientSearch = async (req, res) => {
87    try {
88      const { name } = req.query;
89
90      const clients = await pool.query(
91        "SELECT * FROM client WHERE client_first_name || ' ' || client_last_name ILIKE $1",
92        [`%${name}%`]
93      );
94
95      res.json(clients.rows);
96    } catch (err) {
97      console.error(err.message);
98    }
99  };
100
101 module.exports = { addNewClient, getAllClients, getClientById, updateClientInfo, removeClient, clientSearch };
```

Figure 16 remove client & search

Struktura baze podataka

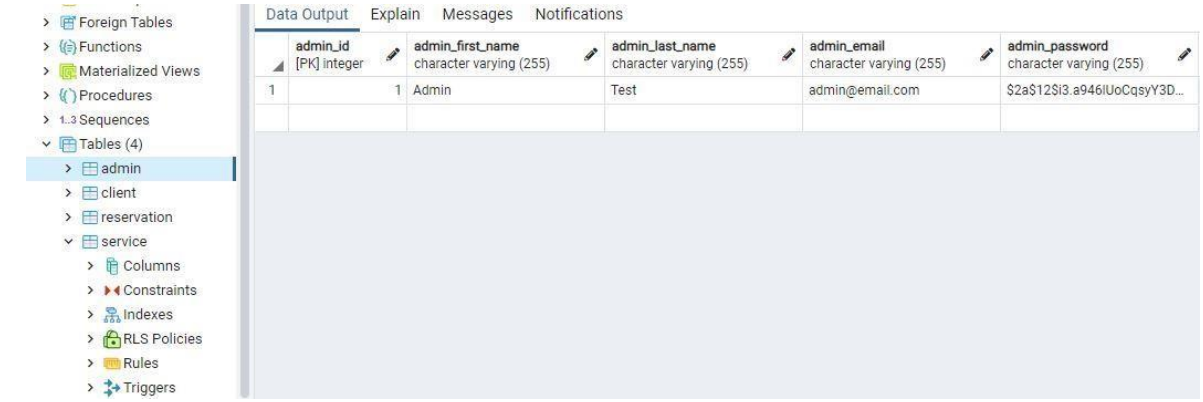

Figure 17 Admin table

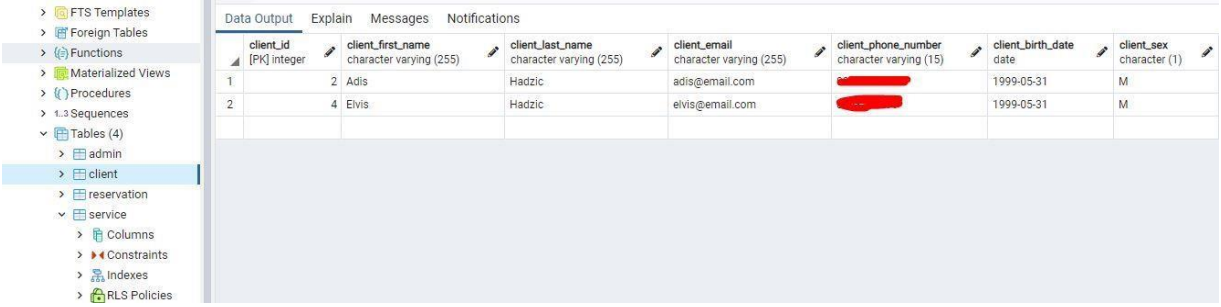

Figure 18 Client table

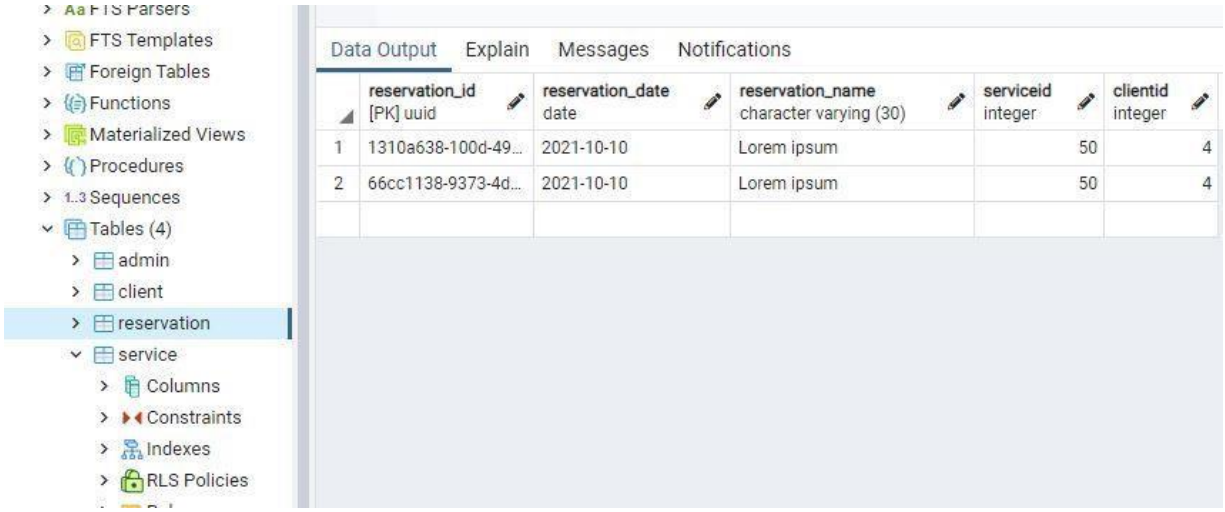

Figure 20 Reservation table

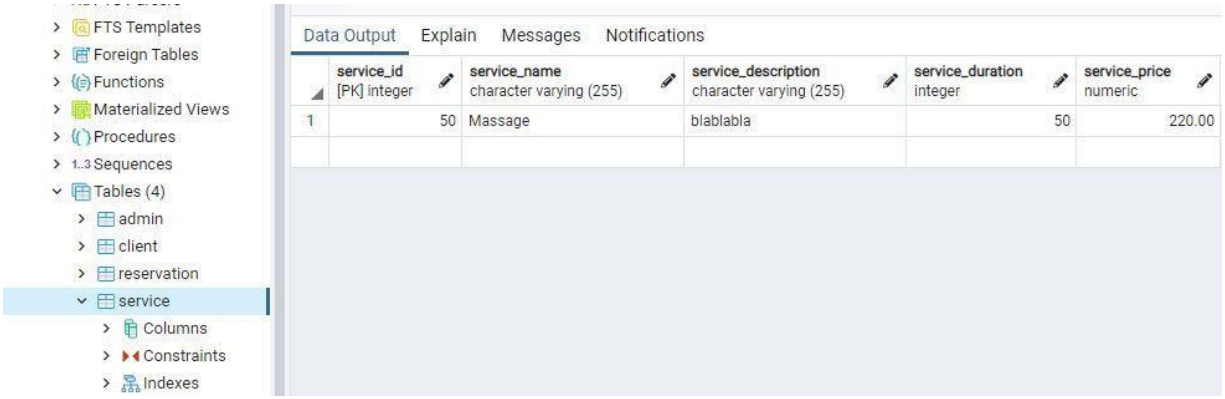

Figure 19 Service table

Reference

Skripte i video predavanja – doc.dr.sc. Nikola Tanković (https://bit.ly/3kj182h)

Postgres Tutorial, https://www.postgresqltutorial.com/

NPM paketi:

- https://www.npmjs.com/package/pg
- https://www.npmjs.com/package/joi
- https://www.npmjs.com/package/express
- https://www.npmjs.com/package/bcrypt
- https://www.npmjs.com/package/jsonwebtoken
- https://www.npmjs.com/package/dotenv
- https://www.npmjs.com/package/cors