Razvoj poslužiteljskog sloja fitnes aplikacije "Readysetgym"

Benković, Luka

Sveučilište Jurja Dobrile u Puli

Fakultet Informatike u Puli

Razvoj poslužiteljskog sloja fitnes aplikacije „ReadySetGym“

Završni rad

Pula, rujan 2024. godine

Sveučilište Jurja Dobrile u Puli
Fakultet Informatike u Puli

Luka Benković

Razvoj poslužiteljskog sloja fitnes aplikacije „ReadySetGym“

Završni rad

JMBAG: 0152208610, redovni student
Studijski smjer: Sveučilišni prijediplomski studij Informatika
Kolegij: Web aplikacije
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: doc. dr. sc. Nikola Tanković

Sažetak

Cilj ovog završnog rada je napraviti poslužiteljski sloj (Backend) i dokumentaciju za fitnes aplikaciju „ReadySetGym“. Klijentski sloj aplikacije dokumentiran je u završnom radu pod nazivom „Razvoj klijentskog dijela web aplikacije za fitness“ koji je izradila Mirna Pušeljić. Web aplikacija je razvijena na način da omogući korisnicima praćenje osobnog napretka na svojem fitnes putovanju, kreiranje vlastitih planova za trening, te povezivanje i real-time komunikaciju između korisnika („GymBros“). Poslužiteljski sloj (Backend) omogućuje korisnicima komunikaciju sa bazom podataka, osiguravajući pritom kako sigurnost korisničkih podataka, tako i pristup tim podacima.

Ključne riječi: MongoDB, Express, Node, REST, Websocket

Abstract

The goal of this bachelor's thesis is to create the server-side layer (Backend) and documentation for the fitness application "ReadySetGym." The client-side layer of the application is documented in the bachelor's thesis titled "Razvoj klijentskog dijela web aplikacije za fitness“ made by Mirna Pušeljić. The web application is designed to allow users to track personal progress on their fitness journey, create their own workout plans, and connect with and communicate in real-time with other users („GymBros“). The server-side layer (Backend) enables users to communicate with the database, ensuring both the security of user data and access to that data.

Keywords: MongoDB, Express, Node, REST, Websocket

Sadržaj

1. Uvod

Razvoj modernih web aplikacija postaje sve kompleksniji, zahtijevajući visoku razinu sigurnosti, pouzdanosti i skalabilnosti, posebno kada su u pitanju aplikacije koje obrađuju osjetljive korisničke podatke. Jedna od takvih aplikacija je i "ReadySetGym", fitnes aplikacija koja korisnicima omogućava praćenje osobnog napretka, kreiranje personaliziranih planova treninga te povezivanje i komunikaciju s drugim korisnicima u realnom vremenu.

Poslužiteljski sloj, poznat i kao backend, ključan je dio ovakve aplikacije jer omogućuje interakciju s bazom podataka, provjeru autentičnosti korisnika, rukovanje zahtjevima iz klijentskog dijela aplikacije te osiguravanje da su podaci korisnika zaštićeni i uvijek dostupni. U ovom radu istražuje se proces razvoja poslužiteljskog sloja za aplikaciju "ReadySetGym" koristeći suvremene tehnologije poput Node.js-a, Express.js-a, i MongoDB-a.

Cilj rada je prikazati kako je backend strukturiran i implementiran, koje su tehnologije korištene, te kako su riješeni ključni izazovi vezani uz sigurnost, performanse i komunikaciju u stvarnom vremenu. Ovaj rad također uključuje detaljnu dokumentaciju koja prati razvojni proces, pružajući uvid u arhitekturu sustava, kao i upute za daljnje održavanje i nadogradnju aplikacije.

Razvijanje poslužiteljskog sloja za "ReadySetGym" predstavlja važan korak u realizaciji funkcionalne i sigurne fitnes aplikacije koja može odgovoriti na potrebe korisnika i pružiti im kvalitetno iskustvo korištenja.

2. Korištene tehnologije

U razvoju poslužiteljskog sloja fitnes aplikacije "ReadySetGym" korišten je niz suvremenih tehnologija koje osiguravaju funkcionalnost, sigurnost i efikasnost sustava.

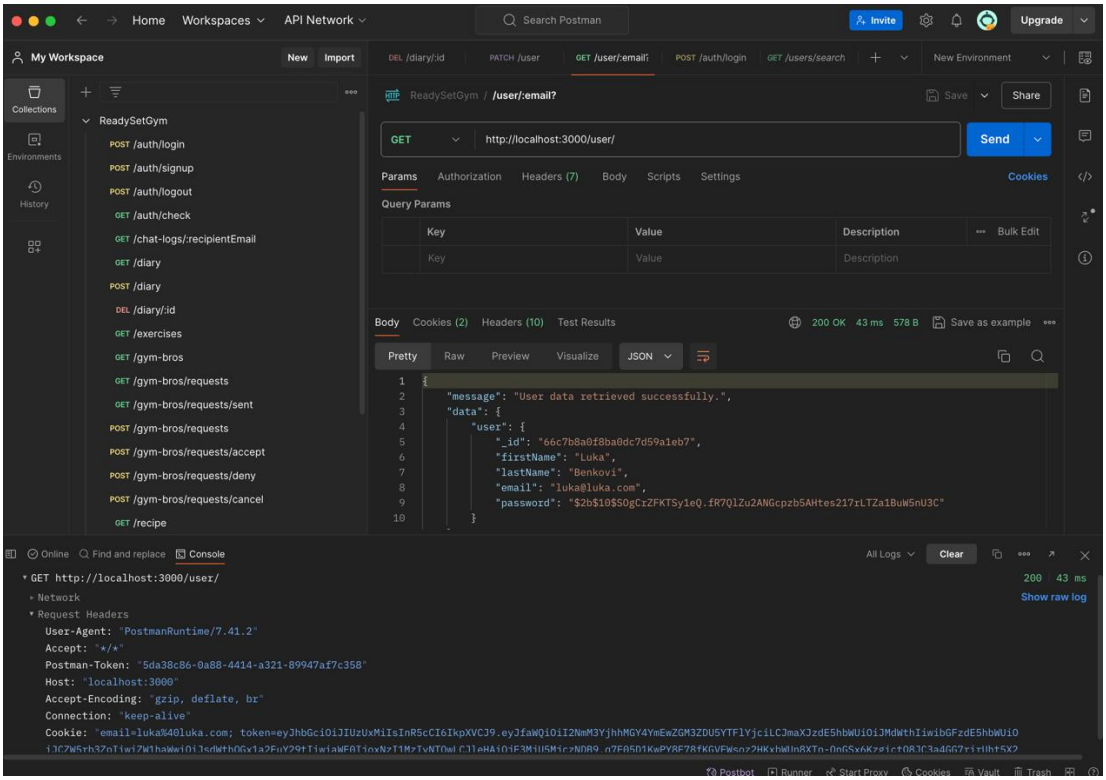
- REST: Arhitektonski stil za dizajn API-ja koji omogućava komunikaciju između klijenta i servera putem standardnih HTTP metoda.
- Postman: Alat za testiranje i dokumentiranje API-ja kroz interaktivno slanje zahtjeva i pregled odgovora.
- Node.js: JavaScript runtime okruženje koje omogućuje izvršavanje JavaScript koda na serveru.
- Express.js: Lagani web framework za Node.js koji pojednostavljuje izgradnju web aplikacija i RESTful API-ja.
- Express-ws: Middleware za Express.js koji dodaje podršku za WebSocket komunikaciju u realnom vremenu.
- MongoDB: NoSQL baza podataka koja pohranjuje podatke u fleksibilnom, nestrukturiranom formatu koristeći dokumente u JSON formatu.
- JWT (jsonwebtoken): Standard za sigurno slanje i verifikaciju podataka između klijenta i servera u obliku JSON Web Tokena.

2.1. REST (Representational State Transfer)

REST je arhitektonski stil koji se koristi za izgradnju skalabilnih i održivih web servisa. U RESTful aplikacijama, resursi su predstavljeni URL-ovima, a klijent i server komuniciraju koristeći standardne HTTP metode kao što su GET, POST, PUT, DELETE, i PATCH. Svaki zahtjev je samodostatan i nosi sve potrebne informacije za njegovo izvršenje, što omogućuje jednostavnu, stateless komunikaciju. [1] REST je idealan za aplikacije poput „ReadySetGym“ jer omogućava efikasnu razmjenu podataka između klijentskog i poslužiteljskog sloja, dok ostaje jednostavan za implementaciju i skaliranje.

2.2. Postman

Postman je popularan alat za testiranje API-ja koji omogućuje programerima da interaktivno šalju HTTP zahtjeve prema serveru, pregledavaju odgovore, te organiziraju i dokumentiraju API zahtjeve u kolekcije. [3] U kontekstu razvoja aplikacije „ReadySetGym“, Postman je korišten za testiranje funkcionalnosti RESTful API-ja tijekom razvoja, omogućujući brzu validaciju ispravnosti zahtjeva, identifikaciju bugova, te dokumentiranje API-ja za buduću upotrebu. Primjer testiranja može se vidjeti na Slici 1. Postman također podržava automatizirano testiranje i izradu skripti, što dodatno olakšava razvoj i održavanje aplikacije.



Slika 1 Primjer testiranja HTTP zahtjeva koristeći Postman

2.3. Node.js

Node.js je JavaScript runtime okruženje temeljeno na V8 engine-u koje omogućuje izvršavanje JavaScript koda izvan web preglednika, na serveru. [4] Node.js je poznat po svojoj asinkronoj, event-driven arhitekturi, što ga čini izuzetno efikasnim u rukovanju velikim brojem istovremenih zahtjeva. U razvoju „ReadySetGym“ aplikacije, Node.js je korišten za izgradnju server-side dijela aplikacije, pružajući visoku performansu i skalabilnost, osobito u real-time komunikaciji između korisnika.

2.4. Express.js

Express.js je minimalistički web framework za Node.js koji pojednostavljuje izgradnju web aplikacija i RESTful API-ja. Express.js nudi bogat skup značajki za rukovanje HTTP zahtjevima, upravljanje rutama, rukovanje middleware funkcijama, i integraciju s različitim bazama podataka. [2] Za aplikaciju „ReadySetGym“, Express.js je korišten za izgradnju server-side logike i API sloja, omogućujući brzu i efikasnu izradu modularnog i skalabilnog backend sustava.

2.5. Express-ws

Express-ws je middleware za Express.js koji dodaje podršku za WebSocket komunikaciju unutar Express aplikacije. [5] WebSocketi omogućuju dvosmjernu komunikaciju između klijenta i servera u realnom vremenu, što je ključno za aplikacije koje zahtijevaju trenutnu razmjenu podataka, poput chat aplikacija ili praćenja uživo. U kontekstu „ReadySetGym“, Express-ws je korišten za implementaciju real-time komunikacije između korisnika, omogućujući brzu i učinkovitu razmjenu poruka i podataka između „GymBros“ korisnika.

2.6. MongoDB

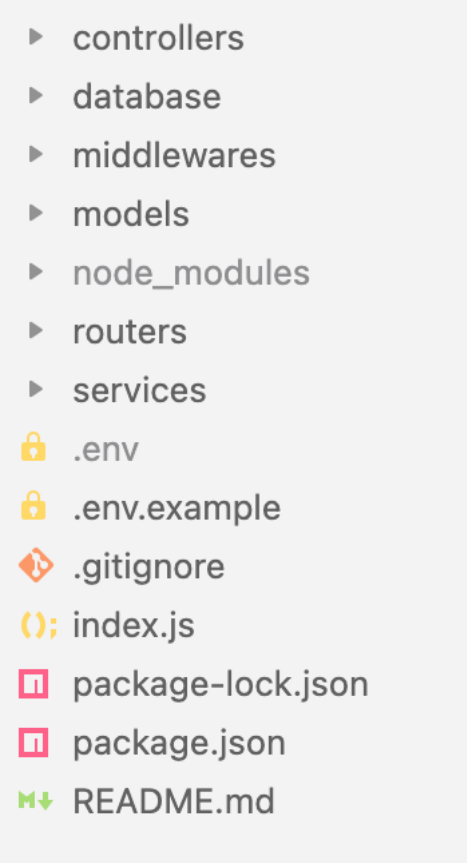
MongoDB je NoSQL baza podataka koja pohranjuje podatke u fleksibilnom, nestrukturiranom formatu koristeći JSON-like dokumente (BSON). MongoDB omogućuje pohranu velikih količina podataka bez potrebe za definiranjem rigidne sheme, što olakšava rad s dinamičnim i promjenjivim podacima. [6] U „ReadySetGym“ aplikaciji, MongoDB je odabran kao primarna baza podataka zbog svoje sposobnosti da skalira, rukuje velikim brojem korisnika, i pohranjuje raznolike podatke, uključujući korisničke profile, planove treninga i povijest aktivnosti.

2.7. JWT (JSON Web Token)

JSON Web Token (JWT) je standard za sigurno slanje i verifikaciju podataka između klijenta i servera. JWT se sastoji od tri dijela: header, payload i signature, te omogućuje autentifikaciju korisnika i autorizaciju pristupa resursima unutar aplikacije. [7] U aplikaciji „ReadySetGym“, JWT je korišten za implementaciju sigurnosnog sloja koji osigurava da samo ovlašteni korisnici mogu pristupiti osjetljivim dijelovima aplikacije i njihovim podacima. JWT omogućuje jednostavnu i sigurnu autentifikaciju koja ne zahtijeva pohranu korisničkih sesija na serveru.

3. Struktura projekta

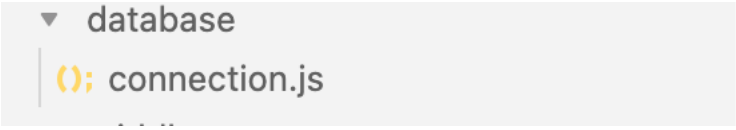
Prilikom razvoja web aplikacija struktura direktorija nam je prilično bitna. Ne samo da bi nama bilo lakše, nego i programerima koji će doći poslije nas i održavati ili nadograđivati aplikaciju. Stoga u glavnoj mapi projekta imamo „controllers“, „models“, „routers“, „services“, „middlewares“, „database“, kako je prikazano na Slici 2.



Slika 2 Struktura projekta

3.1. Database

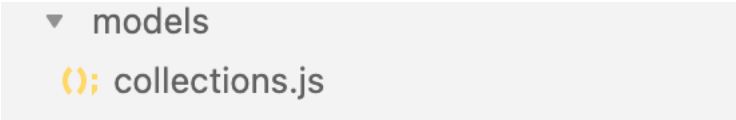
Database direktorij sadrži jednu datoteku, prikazano na Slici 3, koja je zadužena za povezivanje baze podataka. Detaljan opis datoteke u sljedećim poglavljima.



Slika 3 Sadržaj direktorija "database"

3.2. Models

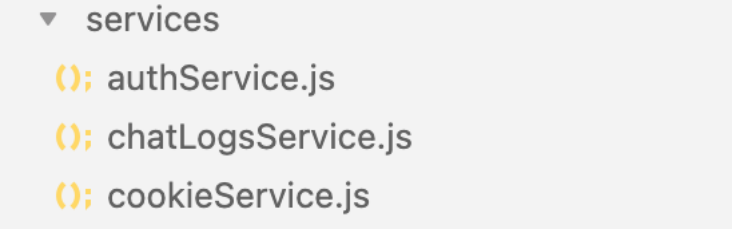
Models direktorij također sadrži jednu datoteku, prikazano na Slici 4, te se nadovezuje na database connection i služi za definiciju strukture baze podataka. Detaljan opis u sljedećim poglavljima.



Slika 4 Sadržaj direktorija "models"

3.3. Services

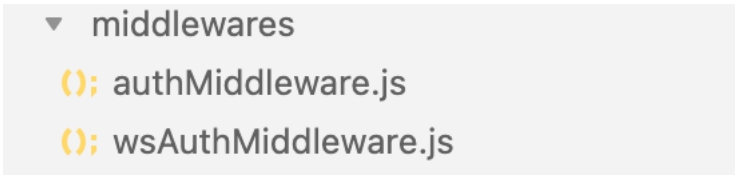
Services direktorij sadrži nekoliko datoteka, prikazano na Slici 5, koje sadrže pomoćne funkcije koje se kasnije koriste u kontrolerima. Pomoćne funkcije radimo jer ih koristimo u više datoteka te kako ne bi morali više puta pisati istu funkciju. Detaljan opis pomoćnih funkcija u sljedećim poglavljima.



Slika 5 Sadržaj direktorija "services"

3.4. Middlewares

Middlewares direktorij sadrži datoteke, prikazane na Slici 6, koje su neizostavne u razvoju web aplikacije u kojoj se koriste osjetljivi podaci, jer nam middleware služi kao barijera između klijentskog dijela i naše baze podataka. Svaki HTTP zahtjev koji dobijemo sa „frontend“-a, mora proći kroz middleware prije nego dopustimo pristup podacima. Više o middleware-ima u sljedećim poglavljima.



Slika 6 Sadržaj direktorija "middlewares"

3.5. Controllers

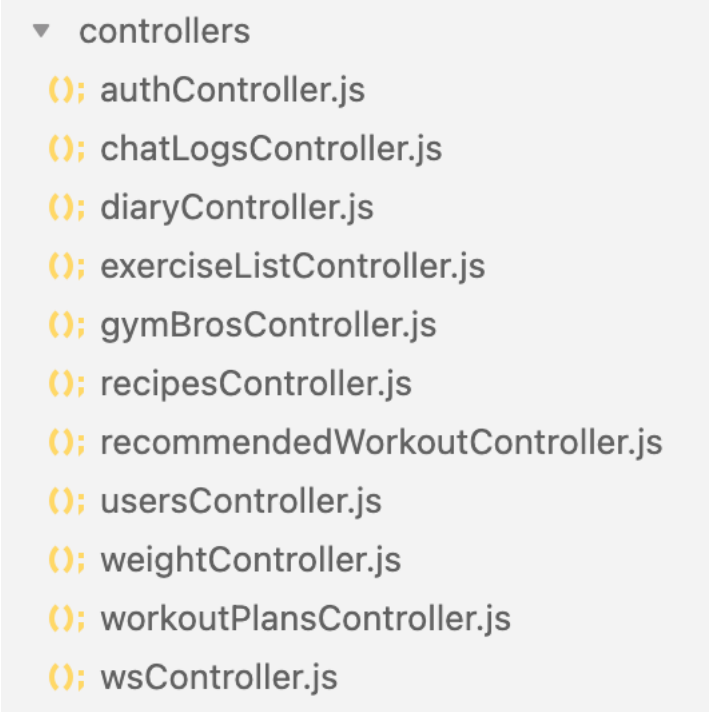
Kontroleri služe kao posrednici između korisničkih zahtjeva i logike aplikacije. Oni upravljaju dolaznim HTTP zahtjevima, komuniciraju s odgovarajućim servisima ili slojevima poslovne logike te vraćaju odgovore natrag klijentu. Glavna funkcija kontrolera je organizirati i usmjeriti protok podataka u aplikaciji.

Ključne uloge kontrolera:

- Obrada korisničkih zahtjeva: Kontroleri primaju zahtjeve iz klijentskog sloja, kao što su GET, POST, PUT, DELETE zahtjevi, i analiziraju ih kako bi razumjeli što korisnik želi.
- Pozivanje servisa i logike poslovanja: Nakon obrade zahtjeva, kontroleri obično pozivaju odgovarajuće servise ili module koji sadrže poslovnu logiku

aplikacije. Oni prosljeđuju podatke servisima, kao što su informacije iz tijela zahtjeva ili parametri rute.

- Komunikacija s bazom podataka: Iako direktna komunikacija s bazom podataka nije uloga kontrolera, oni preko servisa mogu inicirati akcije poput upita u bazu, pohrane podataka ili ažuriranja postojećih zapisa.
- Vraćanje odgovora klijentu: Nakon što su zahtjevi obrađeni, kontroleri formiraju odgovarajući HTTP odgovor, koji može sadržavati podatke, poruku o uspjehu ili grešci, te odgovarajući statusni kod (npr. 200 OK, 404 Not Found, 500 Internal Server Error).
- Rukovanje greškama: Kontroleri također imaju važnu ulogu u rukovanju greškama koje se mogu pojaviti tijekom obrade zahtjeva. Oni osiguravaju da su greške ispravno zabilježene i da je korisniku vraćena odgovarajuća obavijest.



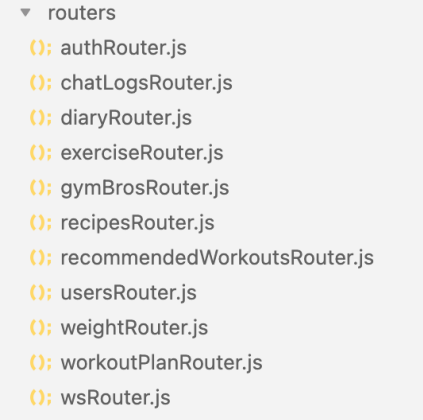
Slika 7 Sadržaj direktorija "controllers"

3.6. Routers

Ruteri služe za usmjeravanje dolaznih HTTP zahtjeva prema odgovarajućim kontrolerima ili funkcijama koje će obraditi te zahtjeve. Oni su ključni dio arhitekture web aplikacija, jer omogućuju organizaciju i strukturiranje rute unutar aplikacije, čineći kod čitljivijim i modularnijim.

Ključne funkcije rutera:

- Usmjeravanje zahtjeva: Ruteri definiraju koje će funkcije ili kontroleri biti pozvani za određene HTTP zahtjeve na temelju URL-a i HTTP metode (npr. GET, POST, PUT, DELETE). Na primjer, zahtjev na URL "/user" može biti usmjeren na funkciju koja dohvaća korisnički profil.
- Modularizacija koda: Korištenjem rutera, aplikacija može biti podijeljena u manje, logične module, gdje se svaki modul bavi određenim dijelom funkcionalnosti. Na primjer, svi zahtjevi vezani za korisnike mogu biti grupirani u jedan router, dok se zahtjevi vezani za planove treninga mogu nalaziti u drugom routeru.
- Održavanje čitljivosti i jednostavnosti: Umjesto da se svi URL-ovi i odgovarajuće funkcije definiraju na jednom mjestu, ruteri omogućuju raspodjelu tih definicija u različite datoteke, što čini kod lakšim za održavanje i proširivanje.
- Primjena middleware-a: Ruteri također omogućuju primjenu middleware funkcija na specifične grupe ruta. Na primjer, možemo dodati autentifikacijski middleware koji će provjeriti je li korisnik prijavljen prije nego što mu se dozvoli pristup određenim rutama.



Slika 8 Sadržaj direktorija "routers"

4. Implementacija poslužiteljskog sloja

U ovom poglavlju ćemo detaljno proći svaku datoteku i objasniti njezinu ulogu u implementaciji poslužiteljskog sloja.

4.1. Postavljanje globalnih varijabli

Kako bi na siguran način uključili tajne podatke kao što su tajna šifra za jwt, connection string za bazu podataka ili API ključeve moramo kreirati .env datoteku u koju ćemo spremiti te varijable koje se mogu kasnije koristi u ostalim datotekama. U svrhu verzioniranja, za što smo koristili GitHub, u slučaju da drugi developer želi pokrenuti lokalno backend napravili smo datoteku .env.example, prikazana na Slici 9, kako bi ostali developeri znali da postoje globalne varijable ili “environment” varijable. U datoteci koja predstavlja primjer glavne .env datoteke samo želimo predstaviti koje varijable moraju biti postavljene za ispravan rad backenda.

.env.example

```
#JWT
JWT_SECRET=

#Database
CONNECTION_STRING=
DATABASE=

#CORS
CORS_ORIGIN=
```

Slika 9 Primjer .env datoteke

4.2. Povezivanje baze podataka

Datoteka za povezivanje baze podataka je kopirana iz dokumentacije od MongoDB-a, uz samo jednu promjenu, a to je dodavanje connection string-a iz .env datoteke umjesto da stoji u connection.js datoteci dostupna svima koji imaju pristup izvornom kodu aplikacije. To nam je bitno za sigurnost podataka jer je connection string jedinstvena kombinacija kojom se poslužitelju baze podataka u ovom slučaju MongoDB-u predstavljamo i tražimo pristup našoj bazi podataka kao administrator.

1. connection.js

```
import dotenv from "dotenv";
dotenv.config();
import { MongoClient } from "mongodb";
const client = new MongoClient(process.env.CONNECTION_STRING);
let conn = null;

try {
    console.log("Establishing connection.");
    conn = await client.connect();
    } catch (e) {
        console.error(e);
    }
let db = conn.db(process.env.DATABASE);

export { client, db };
```

4.3. Definiranje varijabli kolekcija iz baze podataka

MongoDB baza podataka je NoSQL baza podataka i tako za spremanje različitih podataka koriste se “kolekcije” koje sami definiramo. S obzirom da za izvršavanje operacija na kolekcijama moramo definirati na kojoj kolekciji želimo izvršiti operaciju, taj dio ćemo odvojiti u posebnu datoteku te u drugim datotekama mogu samo pozvati varijable.

2. collections.js

```
import { db } from "../database/connection.js";

export const usersCollection = db.collection("users");
export const diaryCollection = db.collection("Diary");
export const exerciseListCollection = db.collection("exerciselist");
export const recipesCollection = db.collection("recipes");
export const recommendedWorkoutsCollection = db.collection(
 "recommendedWorkouts"
);
export const weightCollection = db.collection("weight");
export const workoutPlansCollection = db.collection("WorkoutPlans");
export const gymBrosCollection = db.collection("GymBros");
export const chatLogsCollection = db.collection("ChatLogs");
```

4.4. Pomoćne funkcije (Services)

Pomoćne funkcije, u backend aplikacijama služe za obavljanje ponavljajućih zadataka koji su općenite prirode i mogu se koristiti na različitim mjestima unutar koda. Ove funkcije se često grupiraju u posebne module ili datoteke kako bi se poboljšala organizacija koda i olakšalo ponovno korištenje koda. Iz toga razloga u našoj web aplikaciji postoje tri datoteke sa pomoćnim funkcijama. authService, chatLogsService i cookieService.

4.4.1. Auth Service

Auth Service se sastoji od tri funkcije koje obavljaju različite zadatke. `hashPassword` funkcija koristi se za kodiranje korisničke lozinke kako bi se mogla spremiti na siguran način u korisnikove podatke kako se ne bi spremala u normalno tekstualnom obliku i kako bi osigurali sigurnost podataka. `verifyToken` funkcija koristi JSON Web Token library kako bi se potvrdila ispravnost tokena koji korisnik šalje svakim upitom prema backendu što dodaje sloj sigurnosti. `auth` funkcija obavlja zadatak potvrde postojanja korisnika u bazi podataka te generiranje tokena. Glavna zadaća funkcija je omogućiti korisnicima prijavu u aplikaciju.

3. authService.js

```
export const hashPassword = async (passwordInput) => {
  const hashedPassword = await bcrypt.hash(passwordInput, 10);
  return hashedPassword;
};

export const verifyToken = (accessToken) => {
  try {
    const tokenPayload = jwt.verify(accessToken, process.env.JWT_SECRET);
    return tokenPayload;
  } catch (error) {
    return null;
  }
};

export const auth = async (email, password) => {
  const userData = await usersCollection.findOne({ email: email });
  if (
    userData &&
    userData.password &&
    (await bcrypt.compare(password, userData.password))
  ) {
    delete userData.password;
    const token = jwt.sign(userData, process.env.JWT_SECRET, {
      algorithm: "HS512",
      expiresIn: "1 week",
    });
    return {
      token,
      email: userData.email,
    };
  } else {
    throw new Error("Cannot authenticate");
  }
};
```

4.4.2. Cookie Service

Cookie service se također sastoji od tri funkcije koje dodaju još jedan sloj sigurnosti u aplikaciji na način da dohvaćaju, spremaju i brišu “kolačiće” od korisnika. Za veću sigurnost pristupa bazi podataka koristit ćemo HTTP Only Cookies. Te kolačiće koristimo jer na klijentskom sloju aplikacije (JavaScript) nije moguće pristupiti tim kolačićima. Ova postavka služi kao sigurnosna mjera kako bi se smanjio rizik od krađe kolačića putem napada kao što su Cross-Site Scripting (XSS). `getCookieTokenFromReq` iz korisničkog zahtjeva izvlači neophodne kolačiće u našem slučaju to je tokenCookie i emailCookie. `addAuthCookieToRes` funkcija kao parametre prima token i email te ih dodaje u response prilikom prijave korisnika kako bi mogli biti poslani prilikom poziva sa klijentskog sloja. `removeAuthCookieFromRes` funkcija se koristi prilikom odjave korisnika iz aplikacije kako bi se kolačići obrisali sa klijentskog dijela kako bi onemogućili pristup aplikaciji bez da se korisnik prijavi sa svojim podacima.

4. cookieService.js

```
const TOKEN = "token";
const EMAIL = "email";

export const getCookieTokenFromReq = (req) => {
    const tokenCookie = req.cookies?.[TOKEN];
    const emailCookie = req.cookies?.[EMAIL];
    if (!tokenCookie || !emailCookie) {
        return {};
    }

    return { tokenCookie, emailCookie };
};

export const addAuthCookieToRes = (res, token, email) => {
    res.cookie(TOKEN, token, {
        maxAge: 604800000,
        httpOnly: true,
        secure: true,
        sameSite: "none",
    });
    res.cookie(EMAIL, email, {
        maxAge: 604800000,
        httpOnly: true,
        secure: true,
```

```
        sameSite: "none",
    });
};

export const removeAuthCookieFromRes = (res) => {
    res.clearCookie("token");
    res.clearCookie("email");
};
```

4.4.3. Chat logs Service

Chat logs service sadrži samo jednu funkciju koja obavlja zadatak spremanja poruka iz razgovora sa korisničkog sloja. Glavna primjena funkcije je u websocket controlleru na način da websocket controller sprema poruke u bazu podataka pomoću funkcije `saveMessage` iz servisa i provjerava ako je korisnik kojem je poruka namjenjena spojen na websocket aplikacije i ako je također šalje istu poruku putem websocketa u stvarnom vremenu.

5. chatLogsService.js

```
export const saveMessage = async (sender, recipient, content) => {
    try {
        const existingChat = await chatLogsCollection.findOne({
            participants: { $all: [sender, recipient] },
        });

        if (existingChat) {
            await chatLogsCollection.updateOne(
                { _id: existingChat._id },
                {
                    $push: {
                        messages: {
                            sender,
                            content,
                            timestamp: new Date(),
                        },
                    },
                }
            );
        } else {
            await chatLogsCollection.insertOne({
                participants: [sender, recipient],
                messages: [
                    {
                        sender,
                        content,
                        timestamp: new Date(),
                    },
```

```
                ],
            });
        }
        return true;
    } catch (error) {
        console.error("Error saving message to chatLogs:", error);
        return false;
    }
};
```

4.5. Pomoćni moduli (Middlewares)

Middlewares u poslužiteljskom sloju web aplikacije su jedan od najbitnijih dijelova. Middlewares obavljaju zadatak zaštite ruta na način da provjeravaju korisnikove podatke koje šalje u obliku kolačića kako bi dopustili ili onemogućili pristup rutama te tako i pristup kontrolerima i podacima iz baze podataka. Od iznimne je važnosti dizajnirati middleware da ispravno radi i da je pouzdan način authentifikacije korisnika.

4.5.1. Auth Middleware

Auth middleware ima najveću ulogu u poslužiteljskom sloju na način da se koristi u ruterima prije dopuštanja pristupa tim rutama. Naš auth middleware je dizajniran na način da onemogućava pristup podacima u slučaju da jedan ili oba kolačića koje korisnik pošalje ne odgovara za tog korisnika. U prvom koraku koristimo funkciju koja dohvaća kolačiće iz korisnikovog zahtjeva. U slučaju da jedan od kolačića nedostaje pristup ruti je odbijen sa porukom “Unauthorized”. U slučaju da kolačići postoje prelazi se na drugi korak a to je verifikacija kolačića. Prvo uz pomoć funkcije `verifyToken` provjerava se ispravnost tokena za korisnika, ako je token ispravan prelazi se na verifikaciju emaila te u slučaju da jedan od tih koraka ne prođe verifikaciju pristup rutama i podacima nije dozvoljen. Zbog važnosti middleware-a jako je bitno da middleware radi ispravno.

6. authMiddleware.js

```
export const authMiddleware = (req, res, next) => {
    const { tokenCookie, emailCookie } = getCookieTokenFromReq(req);
    if (!tokenCookie || !emailCookie) {
        return res.status(401).json({ message: "Unauthorized" });
    }
    try {
        const decoded = verifyToken(tokenCookie);
        req.user = decoded;

        if (emailCookie && emailCookie === decoded.email) {
            next();
        } else {
            return res
                .status(401)
                .json({ message: "Email mismatch or not found" });
        }
    } catch (error) {
        return res.status(401).json({ message: "Token is not valid" });
    }
};
```

4.5.2. Websocket Auth Middleware

Websocket auth middleware radi na sličan način kao i prethodni auth middleware. S obzirom da websocket rute radi na drugačiji način od HTTP ruta potrebno je implementirati i drugačiji middleware. Kako na korisničkom sloju nije moguće pristupiti kolačićima, a websocket rute na korisničkom sloju također drugačije funkcioniraju, smo se odlučili na verifikaciju korisničkog emaila kojem se na korisničkom sloju može pristupiti i na taj način poslati u zahtjevu za spajanje na websocket.

7. wsAuthMiddleware.js

```
export const wsAuthMiddleware = async (ws, req, next) => {
    const userEmail = req.query.userEmail;

    if (!userEmail) {
        ws.close(4001, "Unauthorized: No email provided");
        return;
    }

    try {
        const user = await usersCollection.findOne({ email: userEmail });

        if (!user) {
            ws.close(4001, "Unauthorized: Invalid email");
            return;
        }

        req.user = { email: userEmail };
        next();
    } catch (error) {
        console.error("WebSocket authentication error:", error);
        ws.close(4001, "Internal server error");
    }
};
```

4.6. Komponente za obradu zahtjeva

Kontroleri u našem backend-u su podijeljenji ovisno o tome nad kojom kolekcijom u bazi podataka obavljaju podatke, te dva dodatna kontrolera koji imaju svoju posebnu ulogu a to su authController i wsController. Auth controller obavlja ulogu autorizacija korisnika odnosno obavlja zadatke prijave, odjave, registracije i provjere da li je korisnik autoriziran za pristup aplikaciji koji se odvija prilikom prijave te služi korisničkom sloju kao sloj sigurnosti na način da provjeri korisničke podatke prije nego što se korisniku dopusti pristup home pageu. Websocket kontroler obavlja funkcije websocket-a, odnosno osigurava pristup korisnika na websocket što im omogućuje real-time komunikaciju, te ima zadatak rukovanja korisničkim porukama (spremanje u bazu podataka pozivom pomoćne funkcije, prosljeđivanje poruke korisniku kojem je namjenjena).

4.6.1. Auth Controller

Auth controller sastoji se od četiri funkcije koje obavljaju zadatke registracije, prijave, odjave i provjere podataka. Funkcija za registraciju `signup` kriptira korisničku lozinku te stvara novi korisnički dokument u kolekciji "Users" za novog korisnika te vraća poruku da je registracija uspješna. Ako korisnik s istim emailom već postoji vraća poruku da korisnik već postoji i prekida proces registracije. Funkcija za prijavu `login` poziva funkciju auth iz auth servisa za autentifikaciju korisnika i kreiranje jedinstvenog tokena za korisnika te ih vraća na klijetnski sloj u obliku kolačića. Funkcija za odjavu `logout` briše korisničke kolačiće iz response-a i na taj način se korisnik odjavi iz aplikacije. Funkcija `check` se poziva na ruti za provjeru podataka autentifikacije i s obzirom da se koristi authMiddleware prije poziva rute samo vraća poruku "Authenticated".

8. authController.js

```
export const signup = async (req, res) => {
  const userData = req.body;
  const hashedPassword = await bcrypt.hash(userData.password, 10);
  try {
    await usersCollection.insertOne({
      _id: new ObjectId(),
      firstName: userData.firstName,
      lastName: userData.lastName,
      email: userData.email,
      password: hashedPassword,
    });
    return res.json({ message: "Signup successful", data: {} });
  } catch (error) {
    if (error.code === 11000) {
      return res.status(401).json({
        message: "Email already exists. Please choose a different one.",
        data: {},
      });
    } else {
      console.error(`[POST] Signup error: ${error.message}`);
      res.status(500).json({
        message: "Internal server error",
        data: {},
      });
    }
  }
};

export const login = async (req, res) => {
```

```
    const { email, password } = req.body;
    try {
        const authUser = await auth(email, password);
        const token = authUser.token;
        addAuthCookieToRes(res, token, email);
        return res.json({ message: "Login successful", data: {} });
    } catch (error) {
        console.error(`[POST] Login error: ${error.message}`);
        res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};

export const logout = async (req, res) => {
    removeAuthCookieFromRes(res);
    return res.json({
        message: "Logout successful",
        data: {},
    });
};

export const check = async (req, res) => {
    return res.json({
        message: "Authenticated",
        data: {},
    });
};
```

4.6.2. Websocket Controller

Websocket controller sadrži samo jednu funkciju koja upravlja vezama u aplikaciji te omogućuje real-time komunikaciju između korisnika. Kontroler se koristi za rukovanje novim websocket konekcijama, slanje i primanje poruka i praćenje aktivnih konekcija korisnika.

9. wsController.js

```
const activeConnections = {};

export const handleWebSocketConnection = (ws, req) => {
    const userEmail = req.user.email;
    if (!activeConnections[userEmail]) {
        activeConnections[userEmail] = ws;
    }
```

```
    ws.on("message", async (msg) => {
      const parsedMsg = JSON.parse(msg);
      const { recipient, content } = parsedMsg;

      try {
        await saveMessage(userEmail, recipient, content);
        if (activeConnections[recipient]) {
          activeConnections[recipient].send(
            JSON.stringify({
              sender: userEmail,
              content,
              timestamp: new Date(),
            })
          );
        }
      } catch (error) {
        console.error("Error handling message:", error);
      }
    });

    ws.on("close", () => {
      delete activeConnections[userEmail];
    });

    ws.on("error", (error) => {
      console.error("WebSocket error:", error);
    });
};
```

4.6.3. Gym Bros Controller

GymBros kontroler obavlja zadatke prijateljstava u aplikaciji koje smo nazvali “GymBros”. Kontroler se sastoji od sedam funkcija koje obavljaju operacije na kolekciji “GymBros”. Funkcija `getGymBros` dohvaća sve korisnikove prijatelje. Funkcija `getPendingRequests` dohvaća sve zahtjeve koje je korisnik primio od drugih korisnika te korisnik ima opciju prihvatiti ili odbiti zahtjev pomoću funkcija `acceptPendingRequest`, `denyPendingRequest`. Funkcija `savePendingRequest` obavlja funkciju spremanja zahtjeva na čekanju koristeći email pošiljatelja i email primatelje te sprema zahtjev u njihove dokumente u kolekciji. Funkcija `cancelPendingRequest` obavlja funkciju otkazivanja zahtjeva za prijateljstvo odnosno kako funkcija `savePendingRequests` sprema zahtjev u dokumente, funkcija `cancelPendingRequest` briše zahtjev iz njihovih dokumenata. Na kraju funkcija `getSentRequests` dohvaća sve zahtjeve koje je

korisnik poslao drugim korisnicima. Također svaka funkcija ima dio koji osigurava da korisnik ne može sam sebi poslati zahtjev, prihvatiti, odbiti ili otkazati zahtjev prema sam sebi. Također kako bi osigurali atomičnost obavljanja funkcija nad kolekcijama koristili smo transakcije koje nam omogućava MongoDB kao baza podataka što nam osigurava kontinuitet podataka.

10. gymBrosController.js

```
export const getGymBros = async (req, res) => {
  const userEmail = req.cookies.email;

  try {
    const userDoc = await gymBrosCollection.findOne(
      { recipientEmail: userEmail },
      { projection: { acceptedRequests: 1 } }
    );

    if (
      !userDoc ||
      !userDoc.acceptedRequests ||
      userDoc.acceptedRequests.length === 0
    ) {
      return res.json({
        message: "No Gym bros found.",
        data: [],
      });
    }
    return res.json({
      message: "Gym bros fetched successfully.",
      data: userDoc.acceptedRequests,
    });
  } catch (error) {
    console.error(`[GET] Fetch all friends error: ${error.message}`);
    return res.status(500).json({
      message: "Internal server error",
      data: [],
    });
  }
};

export const getPendingRequests = async (req, res) => {
  const recipientEmail = req.cookies.email;

  try {
    const userDoc = await gymBrosCollection.findOne(
      { recipientEmail: recipientEmail },
      { projection: { pendingRequests: 1 } }
    );
```

```
        if (
            !userDoc ||
            !userDoc.pendingRequests ||
            userDoc.pendingRequests.length === 0
        ) {
            return res.json({
                message: "No pending Gym bro requests.",
                data: [],
            });
        }

        return res.json({
            message: "Pending Gym bro requests fetched successfully.",
            data: userDoc.pendingRequests,
        });
    } catch (error) {
        console.error(
            `[GET] Fetch pending Gym bro requests error: ${error.message}`
        );
        return res.status(500).json({
            message: "Internal server error",
            data: [],
        });
    }
};


export const savePendingRequest = async (req, res) => {
    const senderEmail = req.cookies.email;
    const recipientEmail = req.body.email;

    if (senderEmail === recipientEmail) {
        return res.status(400).json({
            message: "You cannot send a request to yourself.",
            data: {},
        });
    }

    try {
        const session = client.startSession();
        session.startTransaction();

        const senderDoc = await gymBrosCollection.findOne(
            { recipientEmail: senderEmail },
            { projection: { pendingRequests: 1, acceptedRequests: 1 } }
        );

        const senderPendingRequests = senderDoc?.pendingRequests || [];
        const senderAcceptedRequests = senderDoc?.acceptedRequests || [];

        if (senderAcceptedRequests.includes(recipientEmail)) {
```

```
            await session.abortTransaction();
            session.endSession();
            return res.status(400).json({
              message:
                "You cannot send a request to someone who is already in your accepted requests.",
              data: {},
            });
          }

          if (senderPendingRequests.includes(recipientEmail)) {
            await session.abortTransaction();
            session.endSession();
            return res.status(400).json({
              message:
                "You cannot send a request to someone who has already sent you a request.",
              data: {},
            });
          }

          await gymBrosCollection.updateOne(
            { recipientEmail: recipientEmail },
            {
              $addToSet: { pendingRequests: senderEmail },
            },
            { upsert: true, session }
          );

          await gymBrosCollection.updateOne(
            { recipientEmail: senderEmail },
            {
              $addToSet: { sentRequests: recipientEmail },
            },
            { upsert: true, session }
          );

          await session.commitTransaction();
          session.endSession();

          return res.json({
            message: "Pending request saved successfully.",
            data: {},
          });
        } catch (error) {
          console.error(`[POST] Gym bros pending error: ${error.message}`);
          return res.status(500).json({
            message: "Internal server error",
            data: {},
          });
        }
      };
```

```
export const acceptPendingRequest = async (req, res) => {
    const recipientEmail = req.cookies.email;
    const senderEmail = req.body.email;

    if (recipientEmail === senderEmail) {
        return res.status(400).json({
            message: "You cannot accept a request from yourself.",
            data: {},
        });
    }

    try {
        const session = client.startSession();
        session.startTransaction();

        const removeResult = await gymBrosCollection.updateOne(
            { recipientEmail: recipientEmail },
            {
                $pull: { pendingRequests: senderEmail },
            },
            { session }
        );

        if (removeResult.modifiedCount === 0) {
            await session.abortTransaction();
            session.endSession();
            return res.status(404).json({
                message: "Gym bro request not found.",
                data: {},
            });
        }

        await gymBrosCollection.updateOne(
            { recipientEmail: recipientEmail },
            {
                $addToSet: { acceptedRequests: senderEmail },
            },
            { session }
        );

        await gymBrosCollection.updateOne(
            { recipientEmail: senderEmail },
            {
                $addToSet: { acceptedRequests: recipientEmail },
            },
            { session }
        );
        await gymBrosCollection.updateOne(
            { recipientEmail: senderEmail },
            {
                $pull: { sentRequests: recipientEmail },
```

```
            },
            { session }
        );

        await session.commitTransaction();
        session.endSession();

        return res.json({
            message: "Gym bro request accepted successfully.",
            data: {},
        });
    } catch (error) {
        console.error(`[POST] Accept Gym bro request error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};

export const denyPendingRequest = async (req, res) => {
    const recipientEmail = req.cookies.email;
    const senderEmail = req.body.email;

    if (recipientEmail === senderEmail) {
        return res.status(400).json({
            message: "You cannot deny a request from yourself.",
            data: {},
        });
    }

    try {
        const session = client.startSession();
        session.startTransaction();

        const removeResult = await gymBrosCollection.updateOne(
            { recipientEmail: recipientEmail },
            {
                $pull: { pendingRequests: senderEmail },
            },
            { session }
        );

        if (removeResult.modifiedCount === 0) {
            await session.abortTransaction();
            session.endSession();
            return res.status(404).json({
                message: "Gym bro request not found.",
                data: {},
            });
        }
```

```
        await gymBrosCollection.updateOne(
            { recipientEmail: senderEmail },
            {
                $pull: { sentRequests: recipientEmail },
            },
            { session }
        );

        await session.commitTransaction();
        session.endSession();

        return res.json({
            message: "Gym bro request denied successfully.",
            data: {},
        });
    } catch (error) {
        console.error(`[POST] Deny Gym bro request error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};

export const cancelPendingRequest = async (req, res) => {
    const senderEmail = req.cookies.email;
    const recipientEmail = req.body.email;

    if (senderEmail === recipientEmail) {
        return res.status(400).json({
            message: "You cannot cancel a request to yourself.",
            data: {},
        });
    }

    try {
        const session = client.startSession();
        session.startTransaction();

        const recipientDoc = await gymBrosCollection.findOne(
            { recipientEmail: recipientEmail },
            { projection: { pendingRequests: 1 } }
        );

        if (
            !recipientDoc ||
            !recipientDoc.pendingRequests ||
            !recipientDoc.pendingRequests.includes(senderEmail)
        ) {
            await session.abortTransaction();
```

```
            session.endSession();
            return res.status(404).json({
                message: "Gym bro request not found.",
                data: {},
            });
        }

        await gymBrosCollection.updateOne(
            { recipientEmail: recipientEmail },
            {
                $pull: { pendingRequests: senderEmail },
            },
            { session }
        );

        await gymBrosCollection.updateOne(
            { recipientEmail: senderEmail },
            {
                $pull: { sentRequests: recipientEmail },
            },
            { session }
        );
        await session.commitTransaction();
        session.endSession();

        return res.json({
            message: "Gym bro request canceled successfully.",
            data: {},
        });
    } catch (error) {
        console.error(`[POST] Cancel Gym bro request error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};
export const getSentRequests = async (req, res) => {
    const userEmail = req.cookies.email;

    try {
        const userDoc = await gymBrosCollection.findOne(
            { recipientEmail: userEmail },
            { projection: { sentRequests: 1 } }
        );
        if (
            !userDoc ||
            !userDoc.sentRequests ||
            userDoc.sentRequests.length === 0
        ) {
            return res.json({
```

```
                message: "No sent Gym bro requests found.",
                data: [],
            });
        }
        return res.json({
            message: "Sent Gym bro requests fetched successfully.",
            data: userDoc.sentRequests,
        });
    } catch (error) {
        console.error(
            `[GET] Fetch sent Gym bro requests error: ${error.message}`
        );
        return res.status(500).json({
            message: "Internal server error",
            data: [],
        });
    }
};
```

4.6.4. Users Controller

Users kontroler se sastoji od tri funkcije koje obavljaju zadatke dohvaćanja, promjene i pretraživanja korisnika u kolekciji “Users”. Funkcija `getUserProfile` dohvaća podatke korisnika koji je trenutno prijavljen u aplikaciju te ih vraća klijentskom dijelu za prikaz korisniku. Funkcija `updateUserProfile` obavlja funkciju promjene korisničkih podatka, te promjenu korisničke slike. Funkcija `searchUsers` obavlja zadatak pretraživanja korisnika kako bi mogli biti pronađeni i dodani na popis “GymBros”.

11. usersController.js

```
export const getUserProfile = async (req, res) => {
    let email = req.cookies.email;
    if (req.params.email) {
        email = req.params.email;
    }
    try {
        const foundUser = await usersCollection.findOne({ email: email });
        if (foundUser) {
            if (req.params.email) {
                const user = {
                    email: foundUser.email,
                    firstName: foundUser.firstName,
                    lastName: foundUser.lastName,
                    imagePath: foundUser.imagePath || null,
                };

                return res.json({
```

```
                message: "Gym bro data retrieved successfully.",
                data: { user },
            });
        } else {
            return res.json({
                message: "User data retrieved successfully.",
                data: { user: foundUser },
            });
        }
    } else {
        return res.status(404).json({
            message: "User not found.",
            data: {},
        });
    }
} catch (error) {
    console.error(`[GET] Users error: ${error.message}`);
    return res.status(500).json({
        message: "Internal server error",
        data: {},
    });
}
};

export const updateUserProfile = async (req, res) => {
    try {
        const userData = req.body;
        const email = req.cookies.email;
        if (userData.new_password) {
            const user = await usersCollection.findOne({ email: email });

            if (
                user &&
                user.password &&
                (await bcrypt.compare(userData.old_password, user.password))
            ) {

                const delta = {};
                if (userData.firstName) {
                    delta.firstName = userData.firstName;
                }
                if (userData.lastName) {
                    delta.lastName = userData.lastName;
                }
                if (userData.new_password) {

                    const newPasswordHash = await bcrypt.hash(
                        userData.new_password,
                        8
                    );
                    delta.password = newPasswordHash;
```

```
            }
            await usersCollection.updateOne(
              { _id: user._id },
              { $set: delta }
            );
          } else {
            return res.json({
              message: "Old password doesn't match.",
              data: {},
            });
          }
        } else {
          const user = await usersCollection.findOne({ email: email });
          if (user && user.password) {
            const delta = { imagePath: userData.imagePath };
            await usersCollection.updateOne(
              { _id: user._id },
              { $set: delta }
            );
          }
        }
        return res.json({
          message: "Profile updated successfully.",
          data: {},
        });
      } catch (error) {
        console.error(`[PATCH] Users error: ${error.message}`);
        return res.status(500).json({
          message: "Internal server error",
          data: {},
        });
      }
    };

    export const searchUsers = async (req, res) => {
      const query = req.params.query;
      try {
        const foundUsers = await usersCollection
          .find({
            $or: [
              { firstName: { $regex: query, $options: "i" } },
              { lastName: { $regex: query, $options: "i" } },
            ],
          })
          .toArray();

        if (foundUsers && foundUsers.length > 0) {
          const users = foundUsers.map(({ _id, password, ...rest }) => rest);

          return res.json({
            message: "Users retrieved successfully.",
```

```
            data: { users },
          });
        } else {
          return res.status(404).json({
            message: "User not found.",
            data: {},
          });
        }
      } catch (error) {
        console.error(`[GET] Users error: ${error.message}`);
        return res.status(500).json({
          message: "Internal server error",
          data: {},
        });
      }
    };
```

4.6.5. Diary Controller

Diary kontroler sadrži tri funkcije koje imaju zadatak spremanja, dohvaćanja i brisanja korisnikovih dnevnika u kolekciju “Diary”. Funkcija `recordDiary` obavlja zadatak spremanja dnevnika u korisnikov dokument u kolekciji. Funkcija `getDiary` obavlja zadatak dohvaćanja korisnikovih dnevnika kako bi se mogle prikazati na korisničkom sloju, dok funkcija `deleteDiary` briše korisnikov dnevnik s id-jem kojeg primi iz korisnikovog zahtjeva.

12. diaryController.js

```
export const recordDiary = async (req, res) => {
  const email = req.cookies.email;
  try {
    const { content, date } = req.body;
    const user = await usersCollection.findOne({
      email: email,
    });

    if (user) {
      await diaryCollection.insertOne({
        email: user.email,
        content,
        date,
      });
      return res.json({
        message: "Diary entry saved successfully.",
        data: {},
      });
    } else {
      return res
        .status(404)
```

```
            .json({ message: "User not found", data: {} });
        }
    } catch (error) {
        console.error(`[POST] Diary error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};

export const getDiary = async (req, res) => {
    const email = req.cookies.email;
    try {
        const diaries = await diaryCollection.find({ email: email }).toArray();
        return res.json({
            message: "User diaries retrieved successfully.",
            data: { diaries },
        });
    } catch (error) {
        console.error(`[GET] Diary error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};

export const deleteDiary = async (req, res) => {
    const email = req.cookies.email;
    try {
        const diaryId = req.params.diaryId;

        const user = await usersCollection.findOne({ email: email });
        if (!user) {
            return res
                .status(404)
                .json({ message: "User not found", data: {} });
        }
        const diaryEntry = await diaryCollection.findOne({
            _id: new ObjectId(diaryId),
            email: user.email,
        });
        if (!diaryEntry) {
            return res
                .status(404)
                .json({ message: "Diary entry not found", data: {} });
        }
        await diaryCollection.deleteOne({ _id: new ObjectId(diaryId) });
        return res.json({
            message: "Diary deleted successfully.",
```

```
            data: { user },
        });
    } catch (error) {
        console.error(`[DELETE] Diary error: ${error.message}`);
        return res
            .status(500)
            .json({ message: "Internal server error", data: {} });
    }
};
```

4.6.6. Recipes Controller

Recipes kontroler također sadrži tri funkcije kao i prethodni Diary kontroler, koje obavljaju iste funkcije spremanja, dohvaćanja i brisanja recepata iz korisničkog dokumenta u kolekciji “Recipes”. Funkcija `recordRecipes` sprema recept u korisnikov dokument u kolekciji, funkcija `getRecipes` dohvaća spremljene recepte korisnika te funkcija `deleteRecipes` briše recept iz korisnikovog dokumenta s id-jem koji primi iz korisnikovog zahtjeva.

13. recipesController.js

```
export const recordRecipes = async (req, res) => {
    const email = req.cookies.email;
    try {
        const { recipe } = req.body;
        const user = await usersCollection.findOne({ email: email });
        if (user) {
            await recipesCollection.insertOne({
                email: user.email,
                recipe,
            });
            return res.json({
                message: "Recipe saved successfully.",
                data: {},
            });
        } else {
            return res
                .status(404)
                .json({ message: "User not found", data: {} });
        }
    } catch (error) {
        console.error(`[POST] Recipes error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};
```

```
export const getRecipes = async (req, res) => {
    const email = req.cookies.email;
    try {
        const recipes = await recipesCollection
            .find({ email: email })
            .toArray();
        return res.json({
            message: "User recipes retrieved successfully.",
            data: { recipes },
        });
    } catch (error) {
        console.error(`[GET] Recipes error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};

export const deleteRecipes = async (req, res) => {
    const email = req.cookies.email;
    const { recipeId } = req.params;
    try {
        const user = await usersCollection.findOne({ email: email });
        if (!user) {
            return res.status(404).json({
                message: "User not found",
                data: {},
            });
        }

        const result = await recipesCollection.deleteOne({
            _id: new ObjectId(recipeId),
            email: user.email,
        });

        if (result.deletedCount > 0) {
            return res.json({
                message: "Recipe deleted successfully.",
                data: {},
            });
        } else {
            return res.status(404).json({
                message: "Recipe not found",
                data: {},
            });
        }
    } catch (error) {
        console.error(`[DELETE] Recipe error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
```

```
            data: {},
        });
    }
};
```

4.6.7. Weight Controller

Weight kontroler također sadrži tri funkcije koje obavljaju iste zadatke kao prethodni Diary kontroler ali na kolekciji “Weight”. Funkcija `recordWeight` sprema korisnikovu težinu u njegov dokument u kolekciji, funkcija `getWeight` dohvaća korisnikovu težinu iz njegovog dokumenta te funkcija `deleteWeight` briše težinu iz dokumenta prema datumu kada je spremljena koji korisnik šalje u zahtjevu.

14. weightController.js

```
export const recordWeight = async (req, res) => {
    const email = req.cookies.email;
    try {
        const { weight } = req.body;
        const user = await usersCollection.findOne({
            email: email,
        });
        if (user) {
            await weightCollection.insertOne({
                email: user.email,
                weight: parseInt(weight),
                date: new Date().getTime(),
            });
            return res.json({
                message: "Weight data saved successfully.",
                data: {},
            });
        } else {
            return res
                .status(404)
                .json({ message: "User not found", data: {} });
        }
    } catch (error) {
        console.error(`[POST] Weight error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};
export const getWeight = async (req, res) => {
    const email = req.cookies.email;
```

```
    try {
      const weights = await weightCollection.find({ email: email }).toArray();
      return res.json({
        message: "Weight data retrieved successfully.",
        data: { weights },
      });
    } catch (error) {
      console.error(`[GET] Weight error: ${error.message}`);
      return res.status(500).json({
        message: "Internal server error",
        data: {},
      });
    }
};
export const deleteWeight = async (req, res) => {
    const email = req.cookies.email;
    const { date } = req.params;
    try {
      await weightCollection.deleteOne({
        email: email,
        date: parseInt(date),
      });
      return res.json({
        message: "Weight data deleted successfully.",
        data: {},
      });
    } catch (error) {
      console.error(`[DELETE] Weight error: ${error.message}`);
      return res.status(500).json({
        message: "Internal server error",
        data: {},
      });
    }
};
```

4.6.8. Workout Plans Controller

Workout plans kontroler sadrži tri funkcije koje obavljaju funkcije spremanja, dohvaćanja i brisanja planova treninga koje korisnik osobno kreira. Funkcije spremaju dokumente u kolekciju “WorkoutPlans” koji sadrže korisnikov email i sve podatke vezane za plan (slika, naziv plana, vježbe). “Funkcija `addNewWorkoutPlan` sprema novi dokument u kolekciju sa svim podacima, funkcija `getUserWorkoutPlans` dohvaća sve korisnikove planove kako bi se prikazali na korisničkom sloju te funkcija `deletePlan` briše plan prema id-ju koji korisnik šalje u zahtjevu.

15. workoutPlansController.js

```
export const addNewWorkoutPlan = async (req, res) => {
    const email = req.cookies.email;
    try {
        const { imageUrl, exercises, name } = req.body;
        if (!email || !name || !exercises || !imageUrl) {
            return res
                .status(400)
                .json({ message: "Missing required fields", data: {} });
        }

        const user = await usersCollection.findOne({ email: email });
        if (!user) {
            return res
                .status(404)
                .json({ message: "User not found", data: {} });
        }
        const result = await workoutPlansCollection.insertOne({
            _id: new ObjectId(),
            email,
            titleImagePath: imageUrl,
            exercisesArray: exercises,
            planName: name,
        });
        if (result.acknowledged) {
            return res
                .status(201)
                .json({ message: "New plan successfully added.", data: {} });
        } else {
            return res
                .status(500)
                .json({ message: "Failed to add new plan.", data: {} });
        }
    } catch (error) {
        console.error(`[POST] Workout plans error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};
export const getUserWorkoutPlans = async (req, res) => {
    const email = req.cookies.email;
    try {
        const workoutPlans = await workoutPlansCollection
            .find({ email: email })
            .toArray();
        return res.json({
            message: "User workout plans retrieved successfully.",
            data: { workoutPlans },
```

```
        });
    } catch (error) {
        console.error(`[GET] Workout plans error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};
export const deletePlan = async (req, res) => {
    const email = req.cookies.email;
    const { planId } = req.params;
    try {
        const user = await usersCollection.findOne({ email: email });
        if (!user) {
            return res.status(404).json({
                message: "User not found",
                data: {},
            });
        }
        const result = await workoutPlansCollection.deleteOne({
            _id: new ObjectId(planId),
            email: user.email,
        });
        if (result.deletedCount > 0) {
            return res.json({
                message: "Plan deleted successfully.",
                data: {},
            });
        } else {
            return res.status(404).json({
                message: "Plan not found",
                data: {},
            });
        }
    } catch (error) {
        console.error(`[DELET] Workout plans error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};
```

4.6.9. Recommended Workouts Controller

Recommended workouts kontroler sadrži samo jednu funkciju koja obavlja funkciju dohvaćanja predloženih planova treninga koje smo ja i kolegica Mirna Pušeljić zajedno složili i koje korisnici mogu samo pregledati. Funkcija `getRecommendedWorkouts`

dohvaća podatke o planu treninga prema nazivu iz kolekcije “RecommendedWorkouts” nakon što korisnik odabere jedan od treninga.

16. recommendedWorkoutsController.js

```
export const getRecommendedWorkouts = async (req, res) => {
  const workoutType = req.params.workoutType;
  try {
    const recommendedWorkouts = await recommendedWorkoutsCollection.findOne(
      { type: workoutType }
    );
    if (recommendedWorkouts) {
      return res.json({
        message: "Recommended workout retrieved successfully.",
        data: { recommendedWorkouts },
      });
    } else {
      res.status(404).json({
        message: "No recommended workouts found for this type",
        data: {},
      });
    }
  } catch (error) {
    console.error(`[GET] Recommended workouts error: ${error.message}`);
    return res
      .status(500)
      .json({ message: "Internal server error", data: {} });
  }
};
```

4.6.10. Exercise List Controller

Exercise list kontroler sadrži samo jednu funkciju koja obavlja zadatak dohvaćanja svih vježbi koje smo ja i kolegica spremili u kolekciju “Exercise List” i koje korisnici mogu odabrati i dodati u svoj plan treninga prilikom kreiranja novog plana. Funkcija `getExercises` dohvaća sve vježbe iz kolekcije koje se onda mogu prikazati na korisničkom sloju.

17. exerciseListController.js

```
export const getExercises = async (req, res) => {
  try {
    const exerciseList = await exerciseListCollection.find({}).toArray();
    return res.json({
      message: "Exercise list retrieved successfully.",
      data: { exerciseList },
```

```
        });
    } catch (error) {
        console.error(`[GET] Exercise list error: ${error.message}`);
        return res.status(500).json({
            message: "Internal server error",
            data: {},
        });
    }
};
```

4.6.11. Chat Logs Controller

Chat logs kontroler sadrži jednu funkciju koja obavlja zadatak dohvaćanja svih poruka koje je korisnik razmjenio sa svojim “GymBro” prilikom ulaska u real-time razgovor kako bi se prikazale na korisničkom sloju. Funkcija `getChatHistory` obavlja taj zadatak dohvaćanja poruka u kolekciji “Chat Logs” prema korisnikovom emailu i emailu prijatelja.

18. chatLogsController.js

```
export const getChatHistory = async (req, res) => {
    const userEmail = req.cookies.email;
    const recipient = req.params.recipientEmail;
    try {
        const chatLog = await chatLogsCollection.findOne({
            participants: { $all: [userEmail, recipient] },
        });
        if (chatLog) {
            return res.json({
                message: "Chat log retrieved successfully.",
                data: { chatLog },
            });
        }
    } catch (error) {
        console.error(`[GET] Chat history error: ${error.message}`);
        res.status(500).json({ message: "Internal server error", data: {} });
    }
};
```

4.7. Komponente za usmjeravanje zahtjeva

Ruteri u našem backendu su organizirani isto kao i kontroleri, a služe za usmjeravanje HTTP zahtjeva prema odgovarajućim kontrolerima. Svaka glavna funkcionalnost aplikacije, kao što su korisničke operacije, upravljanje treninzima, ili rad s porukama, ima svoj zaseban ruter. Ti ruteri povezuju URL putanje s odgovarajućim kontrolerima koji obavljaju specifične zadatke za te resurse. Na primjer, ruter za

korisnike usmjerava zahtjeve vezane uz ažuriranje profila, dok ruter za planove treninga upravlja zahtjevima za kreiranje, ažuriranje i brisanje planova treninga. Pored toga, tu je i ruter za WebSocket, koji osigurava da se zahtjevi vezani uz real-time komunikaciju pravilno prosljeđuju wsControlleru, koji rukuje korisničkim porukama i WebSocket konekcijama. Ruteri na taj način omogućuju jasnu organizaciju aplikacije, jednostavnije održavanje koda, i efikasno rukovanje različitim vrstama HTTP zahtjeva.

4.7.1. Auth Router

Auth ruter usmjerava HTTP zahtjeve za registraciju, prijavu, odjavu i provjeru. U ruteru se koristi authMiddleware gdje je potrebno u odnosu na lokaciju authMiddleware-a u rasporedu ruta, svaka ruta koja se nalazi prije authMiddleware-a neće koristiti authMiddleware, dok svaka ruta koja se nalazi nakon authMiddleware-a zahtjev mora proći kroz authMiddleware prije nego se dopusti pristup toj ruti. U svakoj definiciji rute također se poziva odgovarajuća funkcija iz auth kontrolera.

19. authRouter.js

```
import { Router } from "express";
import { login, logout, signup, check } from "../controllers/authController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";

export const authRouter = () => {
    return Router()
        .post("/auth/login", login)
        .post("/auth/signup", signup)
        .use(authMiddleware)
        .post("/auth/logout", logout)
        .get("/auth/check", check);
};
```

4.7.2. Websocket Router

Websocket ruter ima malo drugačiju sintaksu nego prijašnji ruter jer ne koristi HTTP rute nego posebne ws rute ali radi istu stvar. Kako je u pitanje Websocket ruter umjesto authMiddleware-a koristili smo wsAuthMiddleware te se u ruti poziva funkcija iz websocket kontrolera.

20. wsRouter.js

```
import expressWs from "express-ws";
```

```
import { handleWebSocketConnection } from "../controllers/wsController.js";
import { wsAuthMiddleware } from "../middlewares/wsAuthMiddleware.js";
export const webSocketRouter = (app) => {
    expressWs(app);
    app.ws("/socket", (ws, req) => {
        wsAuthMiddleware(ws, req, () => handleWebSocketConnection(ws, req));
    });
};
```

4.7.3. Gym Bros Router

Gym bros ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje “GymBros”, slanje zahtjeva, dohvaćanja poslanih zahtjeva, prihvaćanje, odbijanje i otkazivanje zahtjeva te dohvaćanje poslanih zahtjeva za prijateljstvo.

21. gymBrosRouter.js

```
import { Router } from "express";
import {
    acceptPendingRequest,
    cancelPendingRequest,
    denyPendingRequest,
    getGymBros,
    getPendingRequests,
    savePendingRequest,
    getSentRequests,
} from "../controllers/gymBrosController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";
export const gymBrosRouter = () => {
    return Router()
        .use(authMiddleware)
        .get("/gym-bros", getGymBros)
        .get("/gym-bros/requests", getPendingRequests)
        .get("/gym-bros/requests/sent", getSentRequests)
        .post("/gym-bros/requests", savePendingRequest)
        .post("/gym-bros/requests/accept", acceptPendingRequest)
        .post("/gym-bros/requests/deny", denyPendingRequest)
        .post("/gym-bros/requests/cancel", cancelPendingRequest);
};
```

4.7.4. Users Router

Users ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje korisnikovog ili prijateljevog profila, pretraživanje drugih korisnika te ažuriranje korisnikovih podataka.

22. usersRouter.js

```
import { Router } from "express";
import {
```

```
    getUserProfile,
    updateUserProfile,
    searchUsers,
} from "../controllers/usersController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";

export const usersRouter = () => {
    return Router()
        .use(authMiddleware)
        .get("/user/:email?", getUserProfile)
        .get("/users/search/:query", searchUsers)
        .patch("/user", updateUserProfile);
};
```

4.7.5. Diary Router

Diary ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje, spremanje i brisanje korisnikovih dnevnika.

23. diaryRouter.js

```
import { Router } from "express";
import {
    recordDiary,
    getDiary,
    deleteDiary,
} from "../controllers/diaryController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";

export const diaryRouter = () => {
    return Router()
        .use(authMiddleware)
        .get("/diary", getDiary)
        .post("/diary", recordDiary)
        .delete("/diary/:diaryId", deleteDiary);
};
```

4.7.6. Recipes Router

Recipes ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje, spremanje i brisanje korisnikovih recepata.

24. recipesRouter.js

```
import { Router } from "express";
import {
    getRecipes,
    recordRecipes,
    deleteRecipes,
} from "../controllers/recipesController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";
```

```
export const recipesRouter = () => {
    return Router()
        .use(authMiddleware)
        .get("/recipe", getRecipes)
        .post("/recipe", recordRecipes)
        .delete("/recipe/:recipeId", deleteRecipes);
};
```

4.7.7. Weight Router

Weight ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje, spremanje i brisanje korisnikovih težina.

25. weightRouter.js

```
import { Router } from "express";
import {
    recordWeight,
    getWeight,
    deleteWeight,
} from "../controllers/weightController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";

export const weightRouter = () => {
    return Router()
        .use(authMiddleware)
        .get("/weight", getWeight)
        .post("/weight", recordWeight)
        .delete("/weight/:date", deleteWeight);
};
```

4.7.8. Workout Plans Router

Workout plans ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje, spremanje i brisanje korisnikovih planova treninga.

26. workoutPlansRouter.js

```
import { Router } from "express";
import {
    addNewWorkoutPlan,
    getUserWorkoutPlans,
    deletePlan,
} from "../controllers/workoutPlansController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";

export const workoutPlanRouter = () => {
    return Router()
        .use(authMiddleware)
        .get("/workout-plan", getUserWorkoutPlans)
```

```
        .post("/workout-plan", addNewWorkoutPlan)
        .delete("/workout-plan/:planId", deletePlan);
};
```

4.7.9. Recommended Workouts Router

Recommended workouts ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje predefiniranih planova treninga.

27. recommendedWorkoutsRouter.js

```
import { Router } from "express";
import { getRecommendedWorkouts } from "../controllers/recommendedWorkoutController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";
export const recommendedWorkoutsRouter = () => {
    return Router()
        .use(authMiddleware)
        .get("/recommendedWorkouts/:workoutType", getRecommendedWorkouts);
};
```

4.7.10. Exercise List Router

Exercise list ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje vježbi.

28. exerciseListRouter.js

```
import { Router } from "express";

import { getExercises } from "../controllers/exerciseListController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";

export const exerciseRouter = () => {
    return Router().use(authMiddleware).get("/exercises", getExercises);
};
```

4.7.11. Chat Logs Router

Chat logs ruter zadužen je za usmjeravanje HTTP zahtjeva za dohvaćanje korisnikovih poruka sa prijateljem.

29. chatLogsRouter.js

```
import { Router } from "express";

import { getChatHistory } from "../controllers/chatLogsController.js";
import { authMiddleware } from "../middlewares/authMiddleware.js";
export const chatLogsRouter = () => {
    return Router()
        .use(authMiddleware)
```

```
        .get("/chat-logs/:recipientEmail", getChatHistory);
};
```

4.8. Glavna pokretačka datoteka

Glavna datoteka za pokretanje poslužiteljskog sloja je “index.js”. Ona je glavna ulazna točka aplikacije. U njoj se inicijalizira Express server, postavljaju ključni middleware-i (za parsiranje JSON-a, kolačića, i CORS), te se povezuju razni routeri za rukovanje specifičnim funkcionalnostima aplikacije, kao što su autentifikacija, korisnički profili, treninzi i real-time komunikacija putem WebSocket-a. Na kraju, datoteka pokreće server na zadanom portu i omogućuje aplikaciji da odgovara na zahtjeve korisnika.

30. index.js

```
import express from "express";

import dotenv from "dotenv";
dotenv.config();
import cors from "cors";
import cookieParser from "cookie-parser";
import { webSocketRouter } from "./routers/wsRouter.js";
import { authRouter } from "./routers/authRouter.js";
import { usersRouter } from "./routers/usersRouter.js";
import { recommendedWorkoutsRouter } from "./routers/recommendedWorkoutsRouter.js";
import { diaryRouter } from "./routers/diaryRouter.js";
import { weightRouter } from "./routers/weightRouter.js";
import { recipesRouter } from "./routers/recipesRouter.js";
import { workoutPlanRouter } from "./routers/workoutPlanRouter.js";
import { exerciseRouter } from "./routers/exerciseRouter.js";
import { gymBrosRouter } from "./routers/gymBrosRouter.js";
import { chatLogsRouter } from "./routers/chatLogsRouter.js";
const app = express();
const port = 3000;
app.use(express.json());
app.use(cookieParser());
app.use(
  cors({
    origin: process.env.CORS_ORIGIN,
    credentials: true,
  })
);
webSocketRouter(app);
app.use("/", authRouter());
app.use("/", usersRouter());
app.use("/", recommendedWorkoutsRouter());
app.use("/", diaryRouter());
app.use("/", weightRouter());
```

```
app.use("/", recipesRouter());
app.use("/", workoutPlanRouter());
app.use("/", exerciseRouter());
app.use("/", gymBrosRouter());
app.use("/", chatLogsRouter());
app.listen(port, () => console.log(`Slušam na portu ${port}!`));
```

5. Zaključak

Izrada poslužiteljskog sloja fitnes aplikacije "ReadySetGym" predstavlja ključan dio razvoja cjelokupne platforme, omogućujući stabilno i sigurno funkcioniranje svih korisničkih funkcionalnosti. Kroz implementaciju REST API-ja koristeći tehnologije kao što su Node.js, Express.js, i MongoDB, stvoren je robustan poslužiteljski sloj koji omogućuje pohranu i upravljanje podacima, autentifikaciju korisnika, te real-time komunikaciju putem WebSocket-a. Modularnost i jasno definirani kontroleri, ruteri, te pomoćne funkcije (services) olakšavaju održavanje i nadogradnju aplikacije, osiguravajući skalabilnost i prilagodljivost budućim potrebama korisnika. Ovaj projekt

nije samo tehnički izazov, već i temelj za daljnji razvoj i unaprjeđenje aplikacije, pružajući korisnicima pouzdan alat za praćenje i poboljšanje njihovog fitnes putovanja.

Literatura

[1] RESTful API Documentation: https://restfulapi.net/

[2] Express.js Guide: https://expressjs.com/en/guide/routing.html

[3] Postman Documentation: https://learning.postman.com/docs/introduction/overview/

[4] Node.js Documentation: https://nodejs.org/docs/latest/api/

[5] Express-ws npm library: https://www.npmjs.com/package/express-ws

[6] MongoDB Guide: https://www.mongodb.com/docs/guides/

[7] JWT Node.js library GitHub repository: https://github.com/auth0/node-jsonwebtoken

Popis slika