Izrada full stack web aplikacije za praćenje osobnih troškova primjenom Spring Boot-a i React-a

Perić, Mihaela

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zadar / Sveučilište u Zadru**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:162:388148

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-01-02**

Repository / Repozitorij:

University of Zadar Institutional Repository

Sveučilište u Zadru

Stručni prijediplomski studij
Informacijske tehnologije

Mihaela Perić

Izrada full stack web aplikacije za praćenje osobnih troškova primjenom Spring Boot-a i React-a

Završni rad

Zadar, 2024.

Sveučilište u Zadru

Stručni prijediplomski studij
Informacijske tehnologije

Izrada full stack web aplikacije za praćenje osobnih troškova primjenom Spring Boot-a i React-a

Završni rad

Studentica:
Mihaela Perić

Mentor:
Doc. dr. sc. Ante Panjkota

Zadar, 2024.

Izjava o akademskoj čestitosti

Ja, **Mihaela Perić,** ovime izjavljujem da je moj **završni** rad pod naslovom **Izrada full stack web aplikacije za praćenje osobnih troškova primjenom Spring Boot-a i React-a** rezultat mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na izvore i radove navedene u bilješkama i popisu literature. Ni jedan dio mojega rada nije napisan na nedopušten način, odnosno nije prepisan iz necitiranih radova i ne krši bilo čija autorska prava.

Izjavljujem da ni jedan dio ovoga rada nije iskorišten u kojem drugom radu pri bilo kojoj drugoj visokoškolskoj, znanstvenoj, obrazovnoj ili inoj ustanovi.

Sadržaj mojega rada u potpunosti odgovara sadržaju obranjenoga i nakon obrane uređenoga rada.

Zadar, 25. studenog 2024.

Sadržaj

1. Uvod ... 1

2. Opis problema i ponuđenog rješenja ... 1

3. Teorijske osnove korištenih tehnologija ... 2

3.1. "Full stack" razvoj web aplikacija ... 2

3.2. Biblioteke i razvojni okviri ... 3

3.3. Backend ... 4

3.3.1. Springov kontekst ... 5

3.3.2. Maven ... 6

3.4. Baza podataka ... 6

3.5. Frontend ... 8

4. Opis ponuđenog rješenja ... 9

4.1. Arhitektura ponuđenog rješenja ... 10

4.2. Konceptualni prikaz ponuđenog rješenja ... 11

4.3. Model baze podataka ... 12

4.4. Serverski dio ... 16

4.4.1. Podatkovni sloj ... 17

4.4.2. Servisni sloj ... 19

4.4.3. Web ili Controller sloj ... 20

4.4.4. Dependency injection i IoC kontejner ... 22

4.4.5. Sigurnost ... 24

4.4.6. Upravljanje greškama na serverskoj strani ... 25

4.5. Korisničko sučelje ... 27

4.5.1. Razvojni okviri ... 32

4.5.2. Povezivanje klijentske i serverske strane ... 32

4.5.3. Omatanje odgovora ... 35

4.6. Automatizacija i raspoređivanje poslovne logike ... 35

4.6.1. Implementacija trigger-a ... 35

4.6.2. @Scheduled metoda 36

5. Rasprava 36

6. Zaključak 38

7. Literatura 39

8. Popis slika 42

Prilozi 45

Sažetak:

Ovaj rad prikazuje i opisuje razvoj full stack web aplikacije "ExpenseTracker". Za razvoj serverskog dijela aplikacije ili backend-a korišten je Spring Boot, a za frontend ili korisničko sučelje koristio se ReactJS. Aplikacija nudi platformu za praćenje osobnih troškova kroz detaljan pregled troškova i ušteđevine u zadanom vremenskom periodu po definiranim kategorijama. Unosom svake transakcije, bio to prihod ili trošak, korisnik može nadodati detaljan opis, datum izvršavanja transakcije i je li ta transakcija dio pretplate. Prednosti ove aplikacije su prikaz troškova i implementacija mjesečnog budžeta što omogućava korisniku kontinuirano praćenje dozvoljenog iznosa za trošenje. Ova aplikacija se dodatno može razvijati i unaprjeđivati, ponajviše na način da se korisniku dozvoli povezivanje s aplikacijom mobilnog bankarstva što bi ubrzalo dodavanje transakcija u aplikaciju.

Ključne riječi: Spring Boot, ReactJS, troškovi, MySQL, full-stack, web aplikacija

1. Uvod

Praćenje i učinkovito upravljanje osobnim financijama mnogima predstavlja značajan izazov. Iako žele štedjeti, nemaju jasan uvid u svoje troškove, što može dovesti do nenamjernog trošenja većih iznosa od planiranog. Kako bi se korisnicima omogućilo bolje upravljanje financijama, osmišljena je i razvijena web aplikacija za praćenje troškova "ExpenseTracker". Ova aplikacija pruža jednostavno korisničko sučelje koje omogućuje korisniku da prati svoje transakcije, pregleda koliko je potrošio ili uštedio u zadanom vremenskom rasponu i postavi si ciljeve štednje. Cilj ove aplikacije je omogućiti korisniku praćenje vlastite potrošnje kako bi efikasnije raspolagao vlastitim financijama. Razvijenom aplikacijom korisnik može jednostavno pratiti svoju potrošnju, uočiti obrasce u svojem ponašanju iz mjeseca u mjesec i vidjeti sve svoje troškove i prihode na jednom mjestu.

Za razvoj aplikacije korišten je Spring Boot za backend tj. serverski dio i React JS za frontend, tj. klijentski dio. Kao baza podataka, korištena je relacijska MySQL baza, a za lakše pisanje SQL upita korišten je Spring Data JPA uz MyBatis razvojni okvir (eng. framework) koji je omogućio jednostavnije pisanje i slanje složenijih upita prema bazi podataka. Osim toga, za komunikaciju s bazom korišten je i ORM Hibernate, a za smanjivanje "boilerplate" koda korištena je Lombok biblioteka.

Za osnovne segmente sigurnosti aplikacije korišten je Spring Security razvojni okvir uz implementaciju JWT tokena za autentikaciju i verifikaciju klijenta. Korisničko sučelje dizajnirano je uz pomoć Material UI i Bootstrap biblioteka.

2. Opis problema i ponuđenog rješenja

U današnjoj ekonomskoj situaciji, mnogima upravljanje osobnim financijama predstavlja problem. Ozbiljnost problema je veća s porastom troškova života, nesigurnosti izazvane inflacijom, osobnom prezaduženošću putem kreditnih kartica ili podizanjem namjenskih, odnosno nenamjenskih kredita. Uz stalnu digitalizaciju i reklame na svakom koraku, sam pojam novca je izgubio svoju vrijednost. Sve više ljudi danas koristi kartice umjesto plaćanja papirnatim novcem, što dodatno mijenja ljudsku percepciju potrošnje i upravljanja novcem [1]. Danas, kako je sve digitalno, tako je i novac postao samo brojka na ekranu.

Mnoge banke danas su implementirale i plaćanje mobitelom, gdje korisnik prisloni mobilni uređaj na POS terminal i plati traženi iznos. Osim toga, danas postoje i banke koje

su potpuno online, tzv. "neobanke"[2] koje nemaju fizičke poslovnice, nego posluju isključivo online (najpoznatiji primjer bi bila tvrtka Revolut [3]). Digitalizacija na ovoj razini, gdje korisniku uopće nije ni potrebna fizička kartica, utječe na kulturu trošenja ljudi. Budući da je fizički novac sve manje korišten i da prevladava upotreba virtualnog novca, može se reći da takav „novac" mnogi ni ne smatraju „pravim" jer nije opipljiv, te ga je stoga veoma jednostavno i trošiti, a da ni ne znamo koliko smo potrošili.

Ovo je jedan od problema koji ova aplikacija rješava. Nakon svake transakcije, korisnik istu može unijeti u aplikaciju, što bi značilo da tada korisnik svjesno vidi koliko je potrošio i koliko mu preostaje od ukupnog iznosa. Na taj način korisnik dobiva jasnu dinamiku i povijest troškova te postaje svjestan svog troškovnog profila čime se stvaraju mogućnosti za povratak vrijednosti značenja novca.

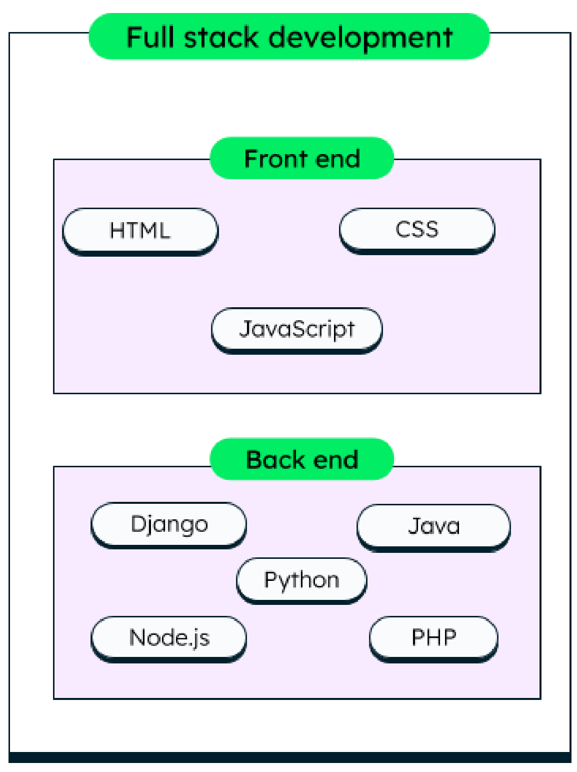
Uz detaljan prikaz trošenja po mjesecu ili po kategoriji, korisnik može pratiti rast ili pad troškova. Tako je lakše uočiti na što se više troši u određenim periodima godine, kada je moguće više štedjeti jer su troškovi u jednom mjesecu bili manji uspoređujući ih s troškovima ostalih mjeseci.

3. Teorijske osnove korištenih tehnologija

U ovoj cjelini opisane su korištene tehnologije za razvoj ove aplikacije i potrebna teorija za razumijevanje ovog rada.

3.1. "Full stack" razvoj web aplikacija

Fullstack razvoj obuhvaća cjelokupni proces razvoja web aplikacija, uključujući i serversku (backend) i klijentsku (frontend) stranu. Kao što se vidi na Slici 1, pod frontend spadaju tehnologije (programski jezici, razvojni okviri) koji su zaduženi za izgradnju korisničkog sučelja poput HTML-a, CSSi JavaScripta ili odgovarajućih razvojnih okvira Prilikom razvoja fullstack web aplikacije, developer mora biti upoznat s velikim brojem klijentskih i serverskih tehnologija kako bi mogao izraditi i povezati sve dijelove aplikacije. Skup tehnologija koje se često zajedno koriste za razvoj aplikacije se naziva tehnološki stog ili eng. *tech stack*. Najpopularniji *tech stack za razvoj web aplikacija* je MEAN što je kratica za MongoDB, Express, Angular i Node.js [4].



Slika 1 Principjelni prikaz Full stack razvoja[5]

3.2. Biblioteke i razvojni okviri

Biblioteke su zbirke unaprijed napisanog koda ili funkcija koje programeri mogu koristiti umjesto pisanja vlastite implementacije [6] i na taj način ubrzali razvoj i izbjegli ponovno pisanje istih dijelova koda. Postoji veliki broj biblioteka za različite programske jezike i svaka nudi rješenje na svoj način za određeni problem. Za razvoj frontend dijela ove aplikacije korišten je ReactJS. ReactJS je Javascript biblioteka koja olakšava razvoj grafičkog sučelja. Korištenjem ReactJS-a developer je u mogućnosti razviti dinamično grafičko sučelje koje se jednostavno spaja na backend preko HTTP zahtjeva.

Uz biblioteke, koje olakšavaju razvoj aplikacija, postoje i razvojni okviri (eng. Application frameworks.) Prema Johnsonu razvojni okvir je „višekratno upotrebljiv dizajn cijelog ili dijela sustava koji je predstavljen skupom apstraktnih klasa i načinom na koji njihove instance međusobno djeluju.[7]" Može se još reći da su razvojni okviri sveobuhvatne softverske strukture koje definiraju arhitekturu i tijek kontrole aplikacije. Za razliku od biblioteka, razvojni okviri upravljaju tokom programa, a razvojni programer popunjava okvir svojim kodom na predviđenim mjestima. Ova inverzija kontrole je ključna razlika između razvojnih okvira i biblioteka. Razvojni okviri često uključuju biblioteke, API-je i dizajnerske obrasce, nudeći sveobuhvatno okruženje za izgradnju aplikacija unutar određene

domene. Dok biblioteke pružaju specifične funkcionalnosti koje razvojni programer može koristiti po potrebi, razvojni okviri određuju cjelokupnu strukturu aplikacije. Za razvoj aplikacija koriste se programski jezici s kojima se mogu koristiti različiti razvojni okviri. Neki od njih su specijalizirani za određene aspekte razvoja poput komunikacije s bazom podataka (npr. Hibernate, MyBatis), i mogu se koristiti zajedno s drugim okvirima koji pokrivaju druge aspekte aplikacije.

3.3. Backend

Razvoj backend-a ili serverskog dijela aplikacije uključuje rad sa serverskim okruženjima i razvijanjem logike same aplikacije. Najpopularniji razvojni okviri za Java programski jezik su Spring, Google Web Toolkit, Grails, Play i mnogi drugi. Spring Boot je razvojni okvir temeljen na Spring-u i nudi dodatne funkcionalnosti. U izradi ove aplikacije korišten je Spring Boot. Odabran je radi svoje jednostavnosti prilikom implementacije i kasnijeg održavanja koda i efikasnih mehanizama naknadnih proširenja funkcionalnosti. Pojednostavljuje razvoj aplikacija jer pruža dodatne funkcionalnosti što smanjuje potrebu za pisanjem velikih količina koda. Uz njega, korišten je i Apache Maven koji također olakšava jednostavno upravljanje potrebnim zavisnostima i bibliotekama. Preko Maven-a, Spring Boot sam preuzima potrebne biblioteke i osigurava kompatibilnost verzija.

Jedna od prednosti Spring Boot-a su upravo zavisnosti, tj. dodaci i integracije koje omogućuju jednostavnije dodavanje potrebnih biblioteka kako bi se proširila funkcionalnost aplikacije. Spring Boot razvojno okruženje je, kao što je već spomenuto, temeljeno na Springu (koji je razvojni okvir za Java programski jezik). Glavna prednost korištenja Spring Boot-a za razvoj aplikacija je ta što je Spring Boot omogućio pisanje aplikacija s veoma malo potrebne konfiguracije. Prije nego što je razvijen Spring Boot, Spring je nudio konfiguraciju kroz XML, dok danas Spring Boot nudi anotacije kao što je @SpringBootApplication [8] iznad BudgetAppApplication klase što je prikazano na Slici 2.

```java
@SpringBootApplication
public class BudgetApplication {

    public static void main(String[] args) {
        SpringApplication.run(BudgetApplication.class, args);
    }

}
```

Slika 2 Primjer upotrebe anotacije iz projekta autorice

Ova anotacija nudi tri značajke (također dostupne kao i zasebne anotacije) koje su uvelike olakšali programiranje u Javi, a to su @EnableAutoConfiguration, @Configuration i @ComponentScan. Anotacija @EnableAutoConfiguration [9] omogućuje autokonfiguraciju Springovog konteksta tako što pokušava pogoditi i konfigurirati beanove koji su nam potrebni. Prema dokumentaciji Spring razvojnog okvira bean je „objekt kojega instancira, sastavlja i na drugi način njime upravlja Spring IoC spremnik." [10]

Ova anotacija provjerava što se nalazi na našem classpath-u i podiže beanove sukladno tome. @ComponentScan anotacija [11] skenira pakete na lokacijama aplikacijske klase i niže te traži @Component anotaciju. Anotacija @Configuration se koristi iznad konfiguracijskih klasa kako bi Spring Boot znao da se tu vjerojatno nalaze beanovi koje treba podignuti u kontekst. U slučaju kada je ova anotacija korištena unutar @SpringBootApplication anotacije, Spring Boot registrira dodatne beanove u kontekst ili uveze dodatne konfiguracijske klase. Ovo je samo jedan od primjera kako je Spring Boot dodatno olakšao programiranje naspram Springa.

3.3.1. Springov kontekst

Springov kontekst (*eng.* Spring Context), koji se koristi u Spring Boot-u, je temeljna komponenta Spring razvojnog okvira koja upravlja kreiranjem i konfiguracijom objekata unutar aplikacije. Temeljni je dio Inversion of Control (IoC) kontejnera. Implementacija IoC principa je također poznata kao dependency injection ili „ubrizgavanje ovisnosti" [12]. Springov kontekst omogućava ubrizgavanje ovisnosti, što znači da *framework* automatski „ubrizgava" potrebne ovisnosti u objekte umjesto da ih programeri ručno kreiraju. Upotrebom „ubrizgavanja ovisnosti" u aplikaciji se onda radi s jednom instancom klase i ima pristup istim vrijednostima objekta, što je zapravo jedan objekt kojem sve klase imaju pristup i nije potrebno da svaka klasa kreira svoju instancu. Osim toga, na ovaj način jasno su izražene ovisnosti objekata unutar aplikacije. Ovo bi se bez IoC kontejnera i

“ubrizgavanja ovisnosti” moglo napraviti da sve klase pristupaju jednoj klasi sa statičkim varijablama i metodama, što je puno kompliciranije jer može doći do nesklada stanja objekta. Na ovaj način Spring Boot je olakšao rad s objektima i održavanje njegovog stanja.

3.3.2. Maven

Prema službenim stranicama, Apache Maven je softverski alat za upravljanje projektima i izgradnju softvera u Javi [13]. Maven projekti su konfigurirani koristeći Project Object Model (POM) unutar pom.xml datoteke. Ova XML datoteka sadrži sve potrebne informacije o projektu i njegovoj konfiguraciji koje Maven koristi za njegovu izgradnju [14]. Unutar pom.xml-a se mogu definirati i zavisnosti, dodaci (eng. plugins) i dodatne konfiguracije potrebne za projekt. Jedna od bitnih značajki Mavena je mogućnost automatskog preuzimanja i upravljanja bibliotekama koje su potrebne za projekt, što olakšava rad na velikim i složenim projektima. Ono što se nalazi u XML datoteci, Maven preuzima sa svog udaljenog repozitorija i sprema na lokalno računalo. Na taj način se olakšava i prijenos projekta jer nije potrebno dodavanje JAR-ova[1] i popisivanje svih potrebnih biblioteka koje trebamo sami skinuti i ubaciti u projekt da bi ga mogli pokrenuti i raditi na njemu.

3.4. Baza podataka

Ovisno o potrebama korisnika, postoje razne vrste baza podataka koje nude različite funkcionalnosti. Neki od osnovnih vrsta su relacijske (SQL) i ne-relacijske (NoSQL) baze podataka. Na Slici 3 opisane su osnovne razlike između relacijskih i nerelacijskih baza podataka. Dok relacijske baze organiziraju podatke u tablice koje sadrže stupce i redove, nerelacijske baze koriste razne vrste spremanja podataka, poput dokumenta, grafa ili spremanja vrijednosti u strukturu ključ-vrijednost. Relacijske baze se još nazivaju i SQL baze jer se za upravljanje koristi SQL (eng. Structured Query Language) pomoću kojeg se preko upita pohranjuju, ažuriraju, pregledavaju ili brišu određeni podaci iz tablice, dok se nerelacijske još nazivaju i NoSQL jer se za upravljanje ovim bazama koriste razni query jezici ovisno o modelu baze.

---

[1] JAR je kratica za Java ARchive. To je format datoteke temeljen na popularnom formatu ZIP datoteke i koristi se za spajanje više datoteka u jednu. (URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html - pristup 15.6.2024.)

RELATIONAL VS NON-RELATIONAL DATABASES

Feature	Relational databases (SQL)	Non-relational databases (NoSQL)
Data structure	• Organize data into tables with rows and columns • Strict schema • Data resides in records and attributes	• Use different data models like ○ document-oriented, ○ key-value, ○ graph, ○ wide-column • Store unstructured data • No fixed schema
Language	Structured Query Language (SQL)	Various query languages depending on the data model
Scalability	• Scale vertically (more computer power to a single server) • Horizontal scaling is challenging and requires additional effort	• Scale horizontally (add more servers) • Share data between servers, decreasing the request-per-second rate in each server
Performance	• Perform well with intensive read/write operations on small to medium datasets • Can suffer when data and user requests grow	• High performance with distributed design • Provide simultaneous access to a large number of users • Store unlimited data sets in various formats
Security	• The integrated structure provides better security • ACID compliance is preferred for applications where database integrity is critical	• Generally weaker security • ACID guarantees are often limited to a single database partition • Some DBMSs offer advanced security features for compliance.
Use cases	Complex software solutions, eCommerce, financial applications	Storing and scaling unstructured data, MVPs for startups, sprint-based Agile development

Slika 3 Relacijske vs ne-relacijske baze podataka[15]

Neke od relacijskih baza podataka su: SQLite, MySQL, MariaDB i PostgreSQL. Iako se za navedene baze koristi SQL, postoje razlike prilikom izvršavanja upita, što znači da neke naredbe su drugačije ovisno o bazi koja se koristi.

Uz relacijske baze podataka, neke od poznatijih nerelacijskih baza podataka su graf baze poput AllegroGraph, ArangoDB, Oracle Graph Database i RedisGraph, koje podatke prikazuju u obliku grafa. Osim graf baza, kao što je već spomenuto, podaci se mogu spremati i u dokumente, a najpoznatija dokument baza podataka je MongoDB, koja koristi dokumente nalik JSON-u (eng. JavaScript Object Notation) što je tekstualni format namijenjen za pohranu i prijenos podataka[16]. Podaci su pohranjeni u dokumentu koji je strukturiran od parova ključ-vrijednost slično kao i kod JSON formata [17].

Za ovu aplikaciju korištena je MySQL baza podataka radi postojanja jasnih relacija između objekata unutar aplikacije. Uočene su pravilne veze između objekata aplikacije, npr. svaki korisnik nakon prijave ima kreiran budžet i kreirane postavke svog korisničkog računa. Jedan budžet može pripadati samo jednom korisniku jer ovisi o transakcijama tog korisnika i mijenja se sukladno njegovim transakcijama. Unosom transakcije, stvaraju se veze između unesene transakcije i korisnika kojem ona pripada – što je veza jedan naprema više, a to znači da jedan korisnik može imati više transakcija, dok jedna transakcija mora pripadati isključivo samo jednom korisniku. Transakcija može, ali ne mora imati poveznicu na pretplatu, što bi značilo da transakcija može ujedno biti i pretplata, no jedna pretplata pripada samo jednoj transakciji. Svaka transakcija pripada nekoj od kategorija i to je veza jedan naprema više jer više transakcija može biti iste kategorije. Kako su podaci pravilno povezani, prikaz ovih veza i praćenje povezanosti objekata jednostavnije je prikazati u pravilnoj strukturi koju nudi relacijska baza podataka.

3.5. Frontend

Da bi korisnik na jednostavan način mogao komunicirati sa serverskom stranom, kreira se grafičko sučelje. Taj dio aplikacije u fullstack arhitekturi naziva se frontend. Razvojem frontenda razvija se grafičko tj. korisničko sučelje preko kojeg se korisniku omogućuje komunikacija sa serverskom stranom. U ranim fazama razvoja web aplikacija, korisničko sučelje se prvenstveno izrađivalo koristeći HTML (eng. Hypertext Markup Language) za strukturu i CSS (eng. Cascading Style Sheets) za definiranje stilova. Ovaj način razvoja bio je ograničen jer aplikacije napisane isključivo u HTML-u i CSS-u nisu bile dinamične niti modularne. Njihove komponente nisu mogle biti višekratno iskorištene, što je dovodilo do ponavljanja koda, otežanog održavanja i smanjenja performansi aplikacija.

Kako bi web aplikacije postale dinamične i jednostavnije za održavanje, počele su se koristiti JavaScript biblioteke i razvojni okviri. Ove tehnologije omogućuju modularnost, gdje se komponente mogu višekratno iskoristiti te se podaci učitavaju dinamično i ažuriraju bez potrebe za osvježavanje cijele stranice. Najpopularnije JavaScript biblioteke i razvojni okviri koji se koriste u modernom frontend razvoju su ReactJS, Vue.js i Angular.

React omogućuje jednostavnu manipulaciju DOM-a (eng. Document Object Model). DOM je API za HTML i XML dokumente i definira logičku strukturu dokumenta te kako se dokumentu pristupa i kako se njime upravlja [18]. ReactDOM je React-ov ključni paket jer povezuje sami React s pravim DOM-om. Omogućuje developerima da pristupaju i

manipuliraju stvarnim elementima DOM-a [19]. Direktna manipulacija DOM-a može biti spora i neefikasna, posebno kod velikih aplikacija s čestim promjenama u korisničkom sučelju. Da bi riješio taj problem, React koristi VirtualDOM. VirtualDOM je lagana kopija stvarnog DOM-a koja se pohranjuje u memoriji. Svaka promjena na sučelju prvo se primjenjuje na VirtualDOM-u koji zatim izračunava minimalan broj stvarnih promjena potrebnih u stvarnom DOM-u [20]. Na taj način, React optimizira ažuriranje sučelja, smanjuje opterećenje na performanse i ubrzava aplikaciju.

4. Opis ponuđenog rješenja

Aplikacija „ExpenseTracker" osmišljena je kao full stack web aplikacija koja je brzo i jednostavno dostupna korisniku putem internet preglednika. Dostupnost aplikacije nije ograničena uređajem, niti zahtjeva određenu platformu kako bi bila dostupna korisniku jer su primijenjeni osnovni principi responzivnog dizajna. Na ovaj način korisnik ima pristup svojim informacijama u bilo kojem trenutku i na bilo kojem uređaju s pristupom internetu.

Neprijavljeni korisnik vidi glavnu stranicu aplikacije. Za interakciju s aplikacijom korisnik mora biti prijavljen. Nakon prijave, korisnik na zaslonu ima kratki pregled nedavnih radnji – nedavno dodane transakcije, koliko je potrošio, a koliko uštedio u tekućem mjesecu i popis ako postoji neka transakcija koja se obnavlja (pretplata) kako bi korisnik mogao predvidjeti budući trošak i otkazati ako nije imao namjenu obnoviti pretplatu.

Korisnik ima dostupne sljedeće radnje:

1. Kreiranje korisničkog računa
2. Prijava i odjava
3. Dodavanje transakcije
4. Uređivanje i brisanje odabrane transakcije
5. Pregled svih transakcija
6. Pregled i brisanje budžeta
7. Izmjena korisničkih postavki
8. Analitički pregled svojeg trošenja

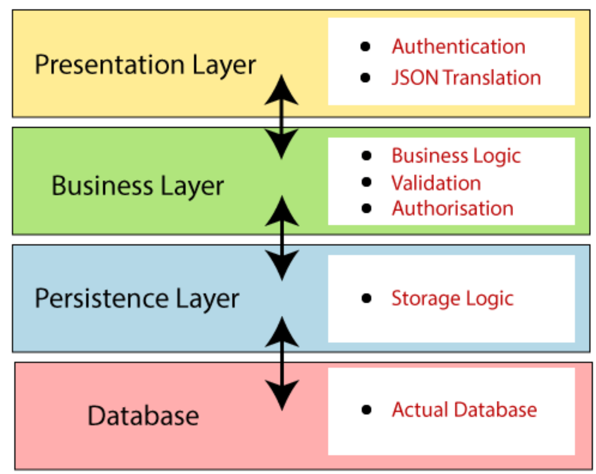
4.1. Arhitektura ponuđenog rješenja

Arhitektura ponuđenog rješenja sastoji se od tri sloja: prezentacijskog sloja, poslovnog ili logičkog i podatkovnog sloja.

Prezentacijski sloj odgovoran je za prikaz podataka korisniku, koje prima putem komunikacije sa serverom. Ova komunikacija je dvosmjerna: prezentacijski sloj prima podatke koje prikazuje korisniku, a korisnik može te podatke mijenjati interakcijom sa sučeljem. Na primjer, korisnik može putem sučelja unositi nove podatke, koji se šalju prema poslovnom sloju na daljnju obradu. Poslovni (logički) sloj posreduje između prezentacijskog i podatkovnog sloja, pružajući funkcionalnost za rukovanje poslovnim pravilima i logikom aplikacije. On obrađuje zahtjeve koji dolaze iz prezentacijskog sloja, šalje upite prema podatkovnom sloju, te nakon toga vraća rezultat prema korisniku. Ovaj sloj nije zadužen samo za dohvaćanje podataka, već i za izvršavanje složenijih operacija nad njima, kao i validaciju i transformaciju podataka prije nego se oni pošalju prema podatkovnom sloju ili prikažu korisniku. Osim toga, omogućuje slanje zahtjeva za spremanje ili ažuriranje podataka.

Podatkovni sloj (*eng*. data or persistence layer) odgovoran je za upravljanje podacima i komunikaciju s bazom podataka. Definira odnose između objekata, omogućava spremanje i dohvaćanje podataka te optimizira rad s bazom. Ovaj sloj također pruža fleksibilnost u načinu na koji se podaci pohranjuju i dohvaćaju, osiguravajući neovisnost aplikacije o vrsti baze podataka koja se koristi. Za rad s bazom podataka koriste se razvojni okviri poput Spring JPA, Hibernatea i MyBatis koji omogućuju jednostavnu implementaciju CRUD operacija (stvaranje, čitanje, ažuriranje, brisanje podataka).

Na Slici 4 vidljiv je blokovski prikaz ovakve arhitekture. Kao što je već spomenuto, a na slici je to dodatno prikazano strjelicama, komunikacija između prezentacijskog i poslovnog sloja, kao i komunikacija između poslovnog i podatkovnog (na slici nazvan *persistence*, no odnosi se na pohranu podataka što je zapravo podatkovni sloj) i komunikacija između podatkovnog sloja i baze podataka, je dvosmjerna te podaci putuju od baze prema korisniku i obrnuto.



Slika 4 Grafički prikaz troslojne arhitekture aplikacija[21]

4.2. Konceptualni prikaz ponuđenog rješenja

U ranijem poglavlju opisane su tehnologije koje su se koristile za razvoj dijelova arhitekture ove aplikacije. Kao što je već spomenuto, aplikacija je razvijena na principu full stack razvoja te uključuje tri komponente; backend, frontend i bazu podataka. Za implementaciju backenda korišten je Java razvojni okvir, Spring Boot-u, dok je za kreiranje grafičkog sučelja korišten ReactJS-u, JavaScript biblioteka koja olakšava razvoj web aplikacija, a za bazu podataka odabrana je MySQL bazu podataka. Preko frontend-a, tj. grafičkog sučelja, korisnik se prijavljuje u aplikaciju ili kreira račun ako nije postojeći korisnik, a to se validira putem backend-a koji provjerava postoji li predano korisničko ime u bazi i je li predana ispravna lozinka.

Konceptualni prikaz je grafički detaljnije objašnjen na Slici 5, gdje se vidi koji dio aplikacije pripada frontend-u, a koji backend-u te za što je koji sloj odgovoran. Iz prikaza je jasno vidljivo da logički sloj služi kao most između prezentacijskog dijela, tj. samog korisnika i baze podataka. Preko njega frontend šalje i prima podatke, a baza podataka zahvaljujući poslovnoj logici koja je implementirana sprema, čita, ažurira ili briše podatke.

Frontend		Backend	
Prezentacijski sloj	Logički sloj		Podatkovni sloj
ReactJS	Spring Boot		MySQL
Prikazivanje korisničkog sučelja i interakcija s korisnikom Dodatni alati: Bootstrap Material UI library	Komunikacija s frontendom Poslovna logika	Komunikacija s bazom podataka Dodatni alati: Spring JPA Hibernate MyBatis	MysSQL Workbench Pohrana podataka, definiranje odnosa i kreiranje trigger-a

Slika 5 Konceptualni prikaz rješenja

Komunikacija između baze i backend-a uspostavljena je preko zavisnosti dodane u pom.xml datoteku: Spring Data JPA-a, MySQL Connector-a i MyBatis razvojnog okvira za složenije upite. Frontend komunicira s backendom preko HTTP zahtjeva tako što konzumira krajnje točke (eng. endpoints) ili putanje iz kontrolera aplikacije. Frontend šalje i prima informacije iz baze preko backend-a. Backend šalje i prima informacije iz baze preko servisnih klasa koje preko repozitorijskih klasa vode računa o spremanju podataka i uzimanju podataka iz baze.

Iako su dio iste aplikacije, frontend i backend su dva odvojena dijela i ako u budućnosti dođe do nekih promjena, npr. promijeni se razvojni okvir jednog od njih, u drugom dijelu će biti minimalnih prilagodbi. Također, backend ne ovisi o korištenoj bazi podataka, samo je bitno da je baza relacijska. Na taj način buduće održavanje same aplikacije je jednostavno, a sve funkcionalnosti aplikacije ne ovise jedna o drugoj što je veoma bitno za otklanjanje bilo kakvih potencijalnih poteškoća ili grešaka.

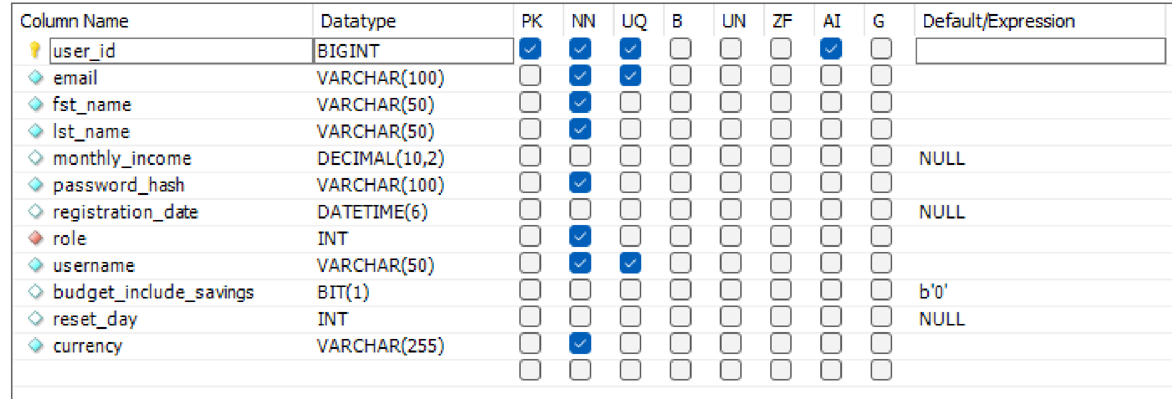
4.3. Model baze podataka

Za ovaj projekt odabrana je relacijska baza. Korištenjem relacijske baze podataka omogućava lakše dohvaćanje podataka preko SQL upita i spremanje samih objekata u bazi preko Spring Data JPA ili MyBatis-a. Obje ovisnosti se koriste s relacijskim bazama. Uz pomoć Hibernate-a, MyBatis i Spring Data JPA-a, spremanje podataka u bazu i rad s njima je vrlo jednostavan. Bazi pristup ima backend dio aplikacije preko URL-a, naziva i lozinke. Ovi podaci zapisani su u aplikacijskoj datoteci application.properties.

Baza ima tablice za glavne objekte - budgets, users, roles, transactions, subscriptions i categories. Svaki user mora imati svoju ulogu, a to može biti USER, ADMIN ili SUPER_ADMIN. Sa userom se povezuju njegove transakcije, a svaka transakcija pripada nekoj kategoriji, može biti pretplata i pripada budžetu.

KORISNIK

Entitet tipa „User" povezan je s tablicom „users" na bazi. Definiran je poljima vidljivima na Slici 6.

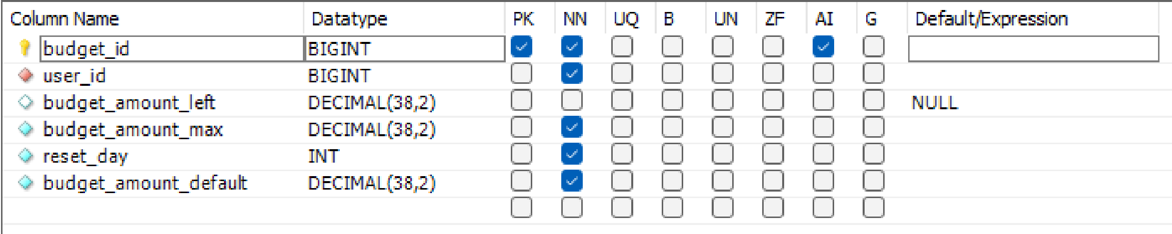


Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
user_id	BIGINT	✓	✓	✓				✓		
email	VARCHAR(100)		✓	✓						
fst_name	VARCHAR(50)		✓							
lst_name	VARCHAR(50)		✓							
monthly_income	DECIMAL(10,2)									NULL
password_hash	VARCHAR(100)		✓							
registration_date	DATETIME(6)									NULL
role	INT		✓							
username	VARCHAR(50)		✓	✓						
budget_include_savings	BIT(1)									b'0'
reset_day	INT									NULL
currency	VARCHAR(255)		✓							

Slika 6 Prikaz tablice Users na bazi

BUDŽET

Entitet tipa „Budget" povezan je s tablicom „budgets" na bazi (Slika 7). Polje user_id je strani ključ što služi kao poveznica između budžeta i njegovog vlasnika te na taj način znamo kojem korisniku pripada koji budžet.

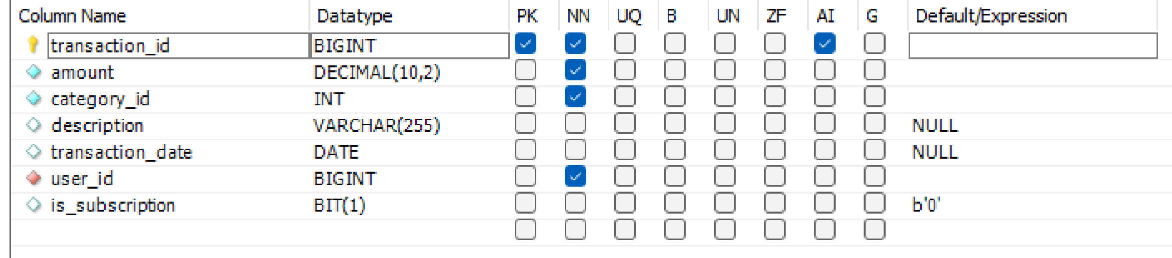


Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
budget_id	BIGINT	✓	✓					✓		
user_id	BIGINT		✓							
budget_amount_left	DECIMAL(38,2)									NULL
budget_amount_max	DECIMAL(38,2)		✓							
reset_day	INT		✓							
budget_amount_default	DECIMAL(38,2)		✓							

Slika 7 Prikaz tablice Budgets na bazi

TRANSAKCIJA

Entitet tipa „Transaction“ povezan je s tablicom „transactions“, a definiran je poljima vidljivima na Slici 8. Polje user_id predstavlja strani ključ i označava kojem korisniku pripada transakcija.

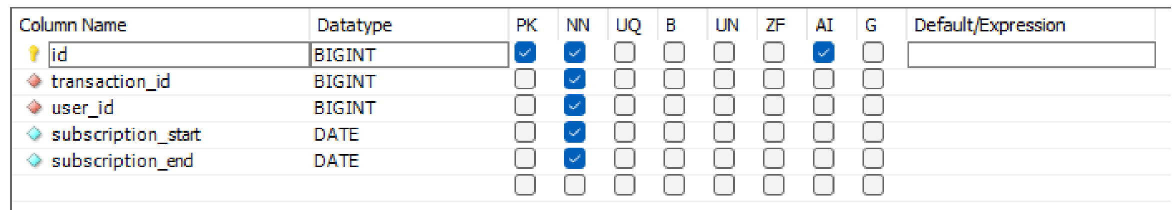


Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
transaction_id	BIGINT	✓	✓					✓		
amount	DECIMAL(10,2)		✓							
category_id	INT		✓							
description	VARCHAR(255)									NULL
transaction_date	DATE									NULL
user_id	BIGINT		✓							
is_subscription	BIT(1)									b'0'

Slika 8 Prikaz tablice Transactions na bazi

PRETPLATA

Entitet tipa „Subscription“ povezan je s tablicom „subscriptions“ i definirana je poljima vidljivima na Slici 9.

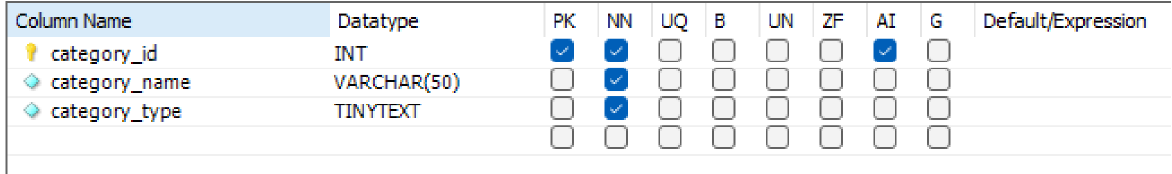


Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	BIGINT	✓	✓					✓		
transaction_id	BIGINT		✓							
user_id	BIGINT		✓							
subscription_start	DATE		✓							
subscription_end	DATE		✓							

Slika 9 Prikaz tablice Subscriptions na bazi

KATEGORIJA

Entitet tipa „Category“ povezan je s tablicom „categories“ (Slika 10).



Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
category_id	INT	✓	✓					✓		
category_name	VARCHAR(50)		✓							
category_type	TINYTEXT		✓							

Slika 10 Prikaz tablice Categories na bazi

POSTAVKE

Entitet tipa „Settings“ povezan je s tablicom „user_settings“ i njezina struktura vidljiva je na Slici 11.

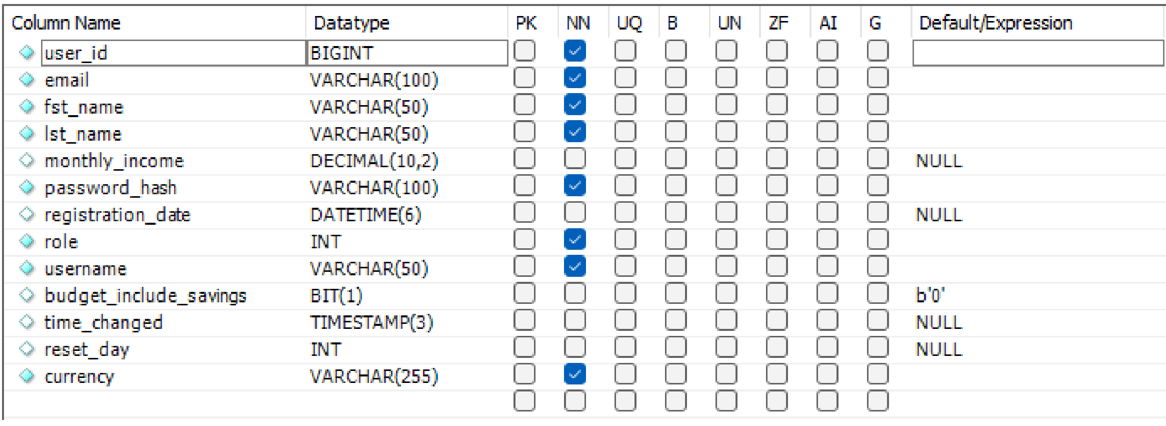
Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
settings_id	BIGINT	✓	✓					✓		
budget_include_savings	BIT(1)									b'0'
max_budget_amount	DECIMAL(38,2)									NULL
monthly_income	DECIMAL(10,2)									NULL
total_savings	DECIMAL(38,2)									NULL
user_id	BIGINT		✓							
username	VARCHAR(50)	✓	✓							
reset_day	INT									NULL

Slika 11 Prikaz User_settings Budgets na bazi

Osim ovih tablica, na bazi se nalaze i tri kronološke tablice koje se pune ovisno o promjenama nad određenim tablicama. To su tablica user_settings_history (Slika 12) koja prati promjene unutar tablice user_settings, tablica users_history (Slika 13) koja prati promjene unutar tablice users i tablica budget_history (Slika 14) koja prati promjene po budžetu korisnika tj. promjene unutar tablice budgets.

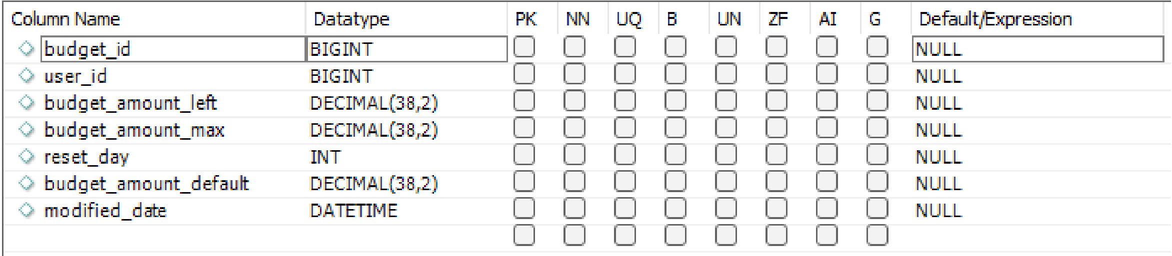
Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
settings_id	BIGINT		✓							
budget_include_savings	BIT(1)									b'0'
max_budget_amount	DECIMAL(38,2)									NULL
monthly_income	DECIMAL(10,2)									NULL
total_savings	DECIMAL(38,2)									NULL
user_id	BIGINT		✓							
username	VARCHAR(50)		✓							
reset_day	INT									NULL
modified_date	DATETIME									NULL

Slika 12 Prikaz tablice User_settings_history na bazi



Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
user_id	BIGINT		✓							
email	VARCHAR(100)		✓							
fst_name	VARCHAR(50)		✓							
lst_name	VARCHAR(50)		✓							
monthly_income	DECIMAL(10,2)									NULL
password_hash	VARCHAR(100)		✓							
registration_date	DATETIME(6)									NULL
role	INT		✓							
username	VARCHAR(50)		✓							
budget_include_savings	BIT(1)									b'0'
time_changed	TIMESTAMP(3)									NULL
reset_day	INT									NULL
currency	VARCHAR(255)		✓							

Slika 13 Prikaz tablice Users_history na bazi



Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
budget_id	BIGINT									NULL
user_id	BIGINT									NULL
budget_amount_left	DECIMAL(38,2)									NULL
budget_amount_max	DECIMAL(38,2)									NULL
reset_day	INT									NULL
budget_amount_default	DECIMAL(38,2)									NULL
modified_date	DATETIME									NULL

Slika 14 Prikaz tablice Budgets_history na bazi

Dijagram entiteti veze uz bazu ovog projekta je dio dodatka APPX. I.

Upiti za dohvaćanje podataka definirani su preko MyBatisa za složenije upite ili koristeći SpringJPA koji putem repozitorija direktno komunicira s bazom, sprema, ažurira, briše i pronalazi podatke prema traženim kriterijima.

4.4. Serverski dio

U serverskom dijelu, koristi se već spomenuti MyBatis [22] za lakše izvršavanje složenih SQL upita. Nudi implementaciju preko XML-a i @Mapper sučelja ili samog @Mapper sučelja. On uklanja većinu JDBC (Java Database Connectivity) koda jer sam ostvaruje konekciju na bazu i sam se brine o njoj preko SqlSessionFactory-a.

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName(Objects.requireNonNull(env.getProperty("spring.datasource.driver-class-name")));
    dataSource.setUrl(env.getProperty("spring.datasource.url"));
    dataSource.setUsername(env.getProperty("spring.datasource.username"));
    dataSource.setPassword(env.getProperty("spring.datasource.password"));
    return dataSource;
}

@Bean
public SqlSessionFactory sqlSessionFactory() throws Exception {
    SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
    sessionFactory.setDataSource(dataSource());
    return sessionFactory.getObject();
}
```

Slika 15 Primjer iz projekta autorice

Na Slici 15 vidimo da podignemo DataSource bean kojem predamo driver, url, username i password, što nam je sve definirano u application.properties datoteci (primjer kako to izgleda u datoteci je vidljiv na Slici 17).

```
9   spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
10  spring.datasource.url=jdbc:mysql://localhost:3306/budget
11  spring.datasource.username=budgetadmin
12  spring.datasource.password=budget123
```

Slika 16 Primjer iz projekta autorice (application.properties datoteka)

Stvaramo SqlSessionFactory bean kojem predajemo Datasource bean kako bi MyBatis znao s kojom bazom radi. U ovom projektu Mybatis se koristi u kombinaciji sa Spring Data JPA i Hibernate-om jer se oni brinu za jednostavnije upita i unose, dok MyBatis omogućuje izvršavanje složenijih upita bez puno muke. Osim ovih alata, korišten je i Lombok [23]. Lombok je biblioteka koja preko anotacija smanjuje količinu tzv. boilerplate koda što je kod koji se ponavlja na mnogo mjesta, poput setter-a i getter-a za svaku klasu ili toString() metode. Takav kod Lombok sam implementira preko anotacija. Za generiranje i provjeru JSON web tokena korišten je JWT.

4.4.1. Podatkovni sloj

Podatkovni sloj brine o podacima i njihovoj pohrani. On povezuje bazu podataka i servisni sloj te se brine o operacijama nad bazom. Ovdje se uglavnom nalaze repozitoriji ili Repository klase za entitete. Ove klase su zapravo sučelja koja implementiraju JpaRepository ili u ovom slučaju CrudRepository jer su nam potrebne samo osnovne funkcionalnosti. JpaRepository proširuje CrudRepository i dodaje mu neke mogućnosti

[24], ali za potrebe ovog projekta dovoljan je i CrudRepository. Neke od metoda su pronalazak objekta po ID-u, ažuriranje objekta, spremanje i brisanje. Samom implementacijom Spring Data JPA repozitorija, ne moramo pisati sve ove metode, nego kreiramo klase i Spring Data JPA automatski definira osnovne metode za nas.

```
9   @Repository
10  public interface UserRepository extends CrudRepository<User, Long> {
11
        10 usages
12      Optional<User> findUserByUsername(String username);
        3 usages
13      Optional<User> findUserByEmail(String email);
14
15  }
```

Slika 17 Prikaz UserRepository klase

Na Slici 19 vidimo kako se implementira CrudRespository tj. Spring Data JPA. Osnovne metode postoje "ispod haube" jer se za njih brine sam Spring Boot, dok developer može sam definirati dodatne metode, kao što su ovdje definirane dvije - findUserByUsername i findUserByEmail. Ove metode Spring Data JPA prepoznaje jer prate konvenciju imenovanja i zna koje sve varijable sadrži predani entitet.

Spring Data JPA se temelji na Jakarta Persistence API-ju (prethodno poznat i kao Java Persistence API) što je zapravo jednostavan razvojni okvir koji se temelji na POJO (Plain Old Java Object) za objekte koji nadžive proces koji ga je kreirao [25], [26] – što bi u ovom slučaju bio user koji je kreiran, a postoji i nakon što je proces gotov ili aplikacija ugašena. Postoji nekoliko implementacija JPA-a, a u ovoj aplikaciji koristi se Hibernate [27], [28]. Hibernate je alat temeljen na ORM-u (Object Relational Mapping) koji mapira entitete aplikacije na njihove pripadajuće tablice u relacijskoj bazi i obrnuto.

U ovom projektu, svaki entitet aplikacije ima svoju Java klasu koja sadrži polja koja su istovjetna kolonama u tablici u bazi. JPA nudi anotaciju @Entity preko koje označavamo klasu kao entitet što znači da se on zapisuje u bazu i čita iz nje. Uz tu anotaciju koristimo i anotaciju @Table kako bi povezali klasu s njezinom tablicom u bazi, a primjer toga vidljiv je na Slici 19, gdje se nad klasom User nalaze anotacije @Entity i @Table, te uz njih vidimo i anotacije za kreiranje getter i setter metoda putem već spomenute biblioteke Lombok, kao i anotacije za kreiranje praznog konstruktora i konstruktora sa svim poljima (isto Lombok).

```java
@Setter
@Getter
@Entity
@Table(name = "users")
@NoArgsConstructor
@AllArgsConstructor
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id", nullable = false)
    private Long userId;

    @Column(name = "fst_name", nullable = false, length = 50)
    private String firstName;
```

Slika 18 Prikaz User klase

Naravno, da bi anotacija @Table mogla funkcionirati moramo imati implementiranu vezu s bazom što nam omogućuje MySQL driver koji smo dodali kao zavisnost u pom.xml datoteci. Uz ove dvije anotacije, vidimo da nad poljima same klase imamo još neke – nad userId poljem, što je ID vrijednost entiteta, nalazi se @Id anotacija koja označava da je to polje identifikator što odgovara primarnom ključu tablice i @GeneratedValue anotacija koja nam upućuje da će se nešto generirati, a unutar zagrada stoji i po kojoj strategiji. U ovoj aplikaciji korištena je strategija GenerationType.IDENTITY što znači da se primarni ključ generira automatskim inkrementom nad kolonom tablice [27]. Da bismo mogli koristiti ovu strategiju generiranja vrijednosti, baza koju koristimo mora imati mogućnost automatskog inkrementa, što neki sustavi za upravljanje bazama možda i ne podržavaju. Anotacija koja povezuje polje entiteta s kolonom tablice je anotacija @Column, gdje kao što vidimo na Slici 19 predajemo naziv kolone iz tablice i neke njene osobnosti. Po tome JPA zapravo zna koje polje pripada kojoj koloni unutar tablice.

4.4.2. Servisni sloj

Servisni sloj se još i naziva i poslovni sloj ili business layer jer sadrži svu poslovnu logiku aplikacije. U ovom dijelu implementirane su servisne klase koje izvršavaju našu poslovnu logiku. Svaki entitet s kojim korisnik može upravljati ima svoju servisnu klasu. Servis se može implementirati na dva načina – preko servisnog sučelja koje ima potrebne metode i servisne klase s konkretnom implementacijom tih metoda ili direktno preko servisne klase koja sadrži implementirane metode. U slučaju različitih implementacija npr. testiranja i produkcije, pristup koji nudi servisno sučelje i konkretnu implementaciju je praktičniji jer podržava polimorfizam. Polimorfizam je sposobnost „podklase“ da definira svoja

jedinstvena ponašanja metoda koje dijeli s „roditeljskom“ klasom. U ovom projektu radi smanjenja obujma klasa i činjenice da nisu potrebne višestruke implementacije, odabran je drugi pristup.

Servisne klase uglavnom se temelje na osnovnim CRUD (Create, Read, Update, Delete) operacijama. Koriste repozitorije kako bi izveli ove osnovne operacije nad podacima. Tako možemo imati createUser(User user) metodu unutar UserService koja pozivom UserRepositoryja omogućuje spremanje novog objekta u bazu. Da bi Spring Boot znao da su ove klase uistinu servisne klase, postoji anotacija @Service. Ova anotacija služi Spring Boot-u da prilikom skeniranja paketa, podigne servisne klase u kontekst i daje drugim klasama na raspolaganje.

4.4.3. Web ili Controller sloj

Web sloj je sloj koji služi kao veza između serverskog dijela aplikacije i korisničkog sučelja. U ovom sloju glavni elementi su kontroleri. Njihova zadaća je primanje klijentskih zahtjeva i njihovo procesiranje uz slanje povratnog odgovora. Kontroler u Spring Boot aplikaciji se označava s anotacijom @RestController ili @Controller. Razlika je u tome što @RestController unutar sebe sadrži i @Controller anotaciju i @ResponseBody anotaciju, što znači da automatski serijalizira povratni objekt metode u HttpResponse objekte [29].

Kontroleri preuzmu zahtjev od klijentskog dijela aplikacije i proslijede ga servisu koji je odgovoran za njegovo obrađivanje. Metode koje kontroler sadrži se nazivaju putanje ili rute (eng. routes). Klijent šalje zahtjev na određenu putanju koja odgovara metodi u kontroleru – što se još naziva i “endpoint” ili krajnja točka.

Ovo se odvija putem HTTP-a (Hypertext Transfer Protocol). HTTP zahtjev sastoji se od nekoliko linija kao što se može vidjeti na Slici 20. U prvoj stoji metoda, putanja i verzija protokola. Nakon toga slijede headers ili zaglavlja koja omogućuju da klijent i server šalju međusobno dodatne informacije. Zadnje je tijelo zahtjeva koje može biti prazno. Između zaglavlja i tijela nalazi se prazna linija.

Slika 19 Objašnjenje HTTP zahtjeva[30]

HTTP metode zahtjeva upućuju na to koje ponašanje želimo postići [31]. S GET metodom radimo dohvat podataka, dok s POST metodom predajemo entitet krajnjoj "meti" ili serveru i očekujemo da on napravi nešto s predanim entitetom. To bi npr. mogla biti registracija novog korisnika gdje predajemo potrebne podatke koje server preuzima putem HTTP POST zahtjeva i dalje obrađuje zahtjev tako da kreira novog korisnika. PUT metoda je slična POST na način da isto predajemo podatke serveru, no PUT ažurira već postojeći objekt s predanim podacima, a ne stvara novi. DELETE metoda briše postojeće podatke sa servera. Postoji još nekoliko HTTP metoda, no u ovom projektu korištene su ove četiri pa ostale nećemo spominjati.

HTTP metode odgovaraju anotacijama u Springu. Na primjer, ako šaljemo GET HTTP zahtjev iznad metode koju pozivamo stajat će @GetMapping.

Unutar HTTP zahtjeva postoje tri načina za slanje podataka ili parametara – path, query i body. Path parametri su dio same putanje (*eng*. path) i dio su URL-a koji se koristi kako bi se identificirali određeni resursi, npr. user/12 gdje je „12" path parametar kojim se definira kojem useru želimo pristupiti. Query parametri se koriste za filtriranje, sortiranje ili prosljeđivanje podataka, a nalaze se nakon upitnika u URL-u i sastoje se od parova ključ-vrijednost, npr. /api/users?user=2 gdje je dio „user=2" query parametar. Višestruki parametri odvajaju se ampersandom (&), npr. /api/users?user=2&name=Ivan. Treći način je preko body parametara. Body parametri se koriste za slanje podataka u tijelu HTTP zahtjeva, najčešće kad se radi o stvaranju ili ažuriranju objekta ili nekog resursa. Za razliku od path i query parametara, body parametri se ne nalaze unutar URL-a. Primjer slanja podataka putem bodyja vidljiv je na Slici 21 ispod.

```
HTTP

1 POST /auth/login HTTP/1.1
2 Host: localhost:8085
3 Content-Type: application/json
4 Content-Length: 57
5
6 {
7     "username": "user",
8     "password": "user1234"
9 }
```

Slika 20 Prikaz poziva endpoint-a iz Postmana

Na Slici 21 je vidljiv poziv metode koja odgovara putanji /auth/login gdje se korisnik prijavljuje sa svojim korisničkim imenom i lozinkom, što vidimo da se nalazi u tijelu zahtjeva. Na Slici 22 prikazan je kod ove metode. Metoda kao ulazni parametar prima @RequestBody tipa LoginDTO preko kojeg se šalju ovi podaci. @RequestBody anotacija se koristi za mapiranje tijela HTTP zahtjeva na zadani Java objekt [32]

```
@PostMapping("/login")
@CrossOrigin(origins = "http://localhost:3000", allowedHeaders = "*")
public ResponseEntity<LoginResponse> loginUser(@RequestBody LoginDTO loginDTO) {
    User authenticatedUser = authService.authenticate(loginDTO);

    String jwtToken = jwtService.generateToken(authenticatedUser);

    LoginResponse loginResponse = new LoginResponse();
    loginResponse.setToken(jwtToken);
    loginResponse.setExpiresIn(jwtService.getExpirationTime());

    return ResponseEntity.ok(loginResponse);
}
```

Slika 21 Prikaz enpoint-a za prijavu korisnika

4.4.4. Dependency injection i IoC kontejner

"Inversion of control" je osnovni princip koji Spring Boot koristi kako bi prebacio brigu o upravljanju komponentama na kontejner. Prije smo spominjali ovaj princip i kako bi se mogao implementirati izvan Springovog okvira, no ispostavilo se da je takav pristup veoma težak za održavanje budući da se o svemu mora brinuti developer. Ono što Spring Boot nudi su anotacije preko kojih on diže objekte ili u ovom slučaju, "beanove" u kontekst i "ubrizgava" ih tamo gdje su potrebni. Na taj način, sve klase koje ovise o drugoj klasi imaju pristup istoj instanci tog objekta. Uz ovaj princip, Spring Boot koristi i obrazac "ubrizgavanja zavisnosti" ili dependency injection kako bi implementirao Inversion of Control princip. Dependency injection je obrazac ponašanja koji omogućuje klasama da

međusobno nisu usko povezane (taj pristup se zove *loose coupling*) i prepušta brigu o upravljanju komponentama kontejneru [33] što znači da Spring Boot podigne sve potrebne komponente u kontekst, ubaci ih tamo gdje su potrebne i brine o njima za vrijeme njihovog životnog ciklusa.

Kako Spring Boot zna što treba podignuti u kontekst i kasnije "ubrizgati"? Spring Boot koristi anotacije, kao što je već spomenuto, i skenira cijeli projekt u potrazi za njima. Također koristi nešto što se zove Java Reflection [34], što znači da ima pristup svemu što se nalazi unutar klase, bilo to privatno (private), zaštićeno (protected), javno (public) ili "default" (access modifiers ili oznake pristupa koje se nalaze iznad definicije klase, metode ili polja). Na taj način Spring Boot zna što sve postoji u projektu i koja klasa je zaslužna za što.

Jedna bitna anotacija koja je zapravo temelj kako Spring Boot "ubrizgava" zavisnosti je @Autowired. Postoje tri načina ubrizgavanja zavisnosti, a to su Setter injection, Field injection i Constructor injection. Setter injection znači da se zavisnost ubacuje preko setter metode. Kao što inače unutar klasa imamo setter metode, ako želimo "ubrizgati" zavisnost o nekom drugom entitetu, iznad metode za postavljanje (set metode) stavimo @Autowired. Field injection, što se ne preporuča, je ubrizgavanje zavisnosti preko polja unutar klase. Tako bi na primjer servisna klasa mogla imati repozitorij objekta kao polje anotiranu s @Autowired za automatsko dodavanje zavisnosti. Ovakav način dodavanja zavisnosti stvara nekoliko problema. Može doći do NullPointerException iznimki ako zavisnosti nisu ispravno inicijalizirane. Nadalje, ova anotacija se ne može koristiti iznad polja označenim s "final" modifikatorom jer Spring Boot radi automatsko ubrizgavanje nakon poziva konstruktora[35], što nije dozvoljeno za polja s modifikatorom final. Ovakva polja moraju biti instancirana prije ili tijekom poziva konstruktora[36] i radi toga je nemoguće koristiti @Autowired iznad njih. "Ubrizgavanje" se mora odviti kroz konstruktor gdje se takva polja mogu ispravno instancirati.

Zadnji način ubrizgavanje zavisnosti je preko konstruktora, što se i preporuča za obvezne ovisnosti [37]. Prilikom podizanja beanova u kontekst, Spring Boot poziva konstruktore svih objekata rekurzivno – što znači da zna hijerarhijski da prvo treba podignuti objekt zavisnosti kako bi ga mogao ubaciti u objekt koji o njemu ovisi – i ubrizgava potrebne zavisnosti. Ova anotacija se i ne mora koristiti kod klasa s jednim konstruktorom jer u tom slučaju dolazi do implicitnog ubrizgavanja putem konstruktora. Čim klasa ima više od jednog konstruktora, anotacija mora biti korištena.

4.4.5. Sigurnost

U ovoj cjelini opisano je kako je implementirana sigurnost na serverskoj strani.

4.4.5.1. Spring Security

Spring Security je razvojni okvir za autentikaciju i kontrolu pristupa koji nudi Spring razvojni okvir. Nudi snažnu podršku za implementaciju sigurnosnih mjera i obuhvaća zaštitu aplikacije kroz autorizaciju i autentikaciju.

4.4.5.2. Autorizacija i autentikacija

Autentikacija je proces kojim se provjerava identitet korisnika, osiguravajući da osoba koja pokušava pristupiti nekom resursu je uistinu ona za koju se predstavlja [38]. U ovom projektu korisnik se autenticira korisničkim imenom i lozinkom, a dalje preko JWT-a (JSON web tokena) kojeg serverski dio aplikacije generira nakon uspješne prijave i validira za potvrdu identiteta korisnika.

Potvrda identiteta korisnika se odvija u JwtAuthentificationFilter klasi, gdje se izvlače podaci koji se koriste za autentikaciju korisnika. HTTP zahtjev sadrži „Authorization" header u kojem se u ovom slučaju nalazi JWT token. „*Authorization*" header se koristi za pružanje vjerodajnica koje autenticiraju *user agent*-a s poslužiteljem, dopuštajući pristup zaštićenim resursima[39]. User agent je „računalni program koji predstavlja osobu, npr. preglednik u web kontekstu.[40]" Sintaksa zahtjeva je vidljiva na Slici 23.

```
const response = await fetch(`http://localhost:8085/transactions/user/addTransaction`, {
    method: 'POST',
    headers: {
        Authorization: `Bearer ${token}`,
        'Content-Type': 'application/json'
    },
    credentials: 'same-origin',
    body: JSON.stringify(transactionToSend)
});
```

Slika 22 Prikaz HTTP zahtjeva s Authorization header-om u React-u

Da bi se korisnik uspješno autenticirao, header mora počinjati s „Bearer". Na taj način znamo da je korisnik izabrao pravu metodu autentikacije, a nakon toga slijedi izvlačenje korisničkog imena iz samog tokena. Token mora biti enkodiran ispravnom tajnom riječi da bi se iz njega moglo izvući korisničkog ime i rok trajanja tokena.

Autorizacija je proces određivanja razine korisničkog pristup nekom resursu [41] nakon što je korisnik autenticiran. Ovim procesom provjerava se što zapravo korisnik smije ili ne smije raditi. U Spring Boot-u, to se postiže određivanjem prava pristupa (eng. roles i permission), U ovoj aplikaciji, korisnik koji se nije prijavio u svoj račun, nema pristup njezinim funkcionalnostima.

Na Slici 24 prikazano je kako se konfigurira sigurnosni filter. Označeni dio metode predstavlja lambda izraz [42] koji definira pravila pristupa za određene rute unutar aplikacije. Metoda .authorizeHttpRequests koristi lambda izraz kako bi se konfigurirala sigurnost različitih putanja. Koristeći .requestMatchers, omogućuje se da se specificiraju koje su putanje (i podputanje) dostupne svim, a koje samo autenticiranim korisnicima. Ovdje vidimo da su putanje koje počinju s /users/, /transactions/, /auth/, /categories/, /budget/ i /analytics/, kao i sve njihova podputanje dostupna samo autenticiranim korisnicima. To nam omogućuje metoda .authenticated() koja označava da korisnici moraju biti prijavljeni kako bi pristupili tim resursima. Neautenticiranom korisniku bit će neomogućen pristup resursima na ovim putanjama, što će obično rezultirati preusmjeravanjem na početnu stranicu ili vraćanjem HTTP odgovora 401 (UNAUTHORIZED). Odgovor koji će se prikazati korisniku ovisi o tome kako se upravlja ovakvim greškama na klijentskoj strani.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(
                    csrf -> csrf
                            .ignoringRequestMatchers( ...patterns: "/users/**"
                                    , "/transactions/**", "/auth/**", "/categories/**",
                                    "/budget/**", "/analytics/**")
            )
            .authorizeHttpRequests(authorize -> authorize
                            .requestMatchers("/users/**", "/transactions/**", "/auth/**",
                                    "/categories/**", "/budget/**", "/analytics/**")
                    .authenticated()
            ).sessionManagement(sessionManagement -> sessionManagement
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authenticationProvider(authenticationProvider)
            .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class)
            .build();
}
```

Slika 23 Prikaz prava pristupa

4.4.6. Upravljanje greškama na serverskoj strani

Za upravljanje greškama na serverskoj strani koristi se anotacija @ControllerAdvice [43]. U ovom slučaju, ova anotacija se koristi iznad GlobalExceptionHandler klase koja upravlja svim greškama tj. iznimkama koje se mogu pojaviti.

```java
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<String> handleResourceNotFound(ResourceNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(UnauthorizedException.class)
    public ResponseEntity<String> handleUnauthorizedException(UnauthorizedException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.UNAUTHORIZED);
    }

    @ExceptionHandler(InvalidTokenException.class)
    public ResponseEntity<String> handleInvalidTokenException(InvalidTokenException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGeneralException(Exception ex) {
        return new ResponseEntity<>( body: "Server error", HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Slika 24 Prikaz @ControllerAdvice klase

Stvorene su tri klase koje proširuju RuntimeException [44] klasu: InvalidTokenException, ResourceNotFoundException i UnauthorizedException. Svaka od ovih klasa predstavlja jednu metodu unutar GlobalExceptionHandler klase (Slika 25). ResponseEntityNotFound se koristi za upravljanje greškama kada servisne klase vrate prazan odgovor tj. da nema resursa kojeg korisnik traži. U tom slučaju vraća se natrag ova iznimka s porukom da resurs nije pronađen i na frontend dijelu aplikacije se dalje raspolaže s tim odgovorom, npr. poruka se prikazuje korisniku na sučelju.

Za upravljanje pristupom imamo InvalidTokenException kad je predani token neispravan ili istekao i UnauthorizedException kad je token ispravan, ali ne postoji korisnik s tim korisničkim imenom.

```java
@PostMapping(value = "/user/addTransaction",
        produces = "application/json",
        consumes = "application/json")
@PreAuthorize("isAuthenticated()")
@CrossOrigin(origins = "http://localhost:3000", allowedHeaders = "*")
public ResponseEntity<?> addTransaction(@RequestHeader(value="Authorization") String token,
                                        @RequestBody AddTransactionRequest addTransactionRequest){

    String username = authService.extractUsernameFromToken(token);
    User user = authService.validateAndGetUser(username);

    transactionService.addTransaction(user, addTransactionRequest);
    return ResponseEntity.status(HttpStatus.CREATED).build();
}
```

Slika 25 Prikaz endpoint-a za dodavanje transakcije

Na Slici 26 se vidi primjer kako se upravlja greškama u jednoj od metoda iz TransactionController klase. Dio koda koji bi se ponavljao u svakoj metodi izvučen je u zasebne dvije metode u AuthService klasi što je prikazano na Slici 27.

```
81      public String extractUsernameFromToken(String token) {
82          if (token == null || token.isEmpty()) {
83              throw new UnauthorizedException("No token.");
84          }
85
86          String username = ExtractJWT.payloadJWTExtraction(token,  extraction: "sub");
87          if (username == null || username.isEmpty()) {
88              throw new InvalidTokenException("Invalid token.");
89          }
90
91          return username;
92      }
93
        14 usages
94      public User validateAndGetUser(String username) {
95          return userRepository.findUserByUsername(username)
96                  .orElseThrow(() -> new ResourceNotFoundException("No users with " + username));
97      }
98
99  }
```

Slika 26 Prikaz ponavljajućeg koda koji se izvukao u zasebne metode

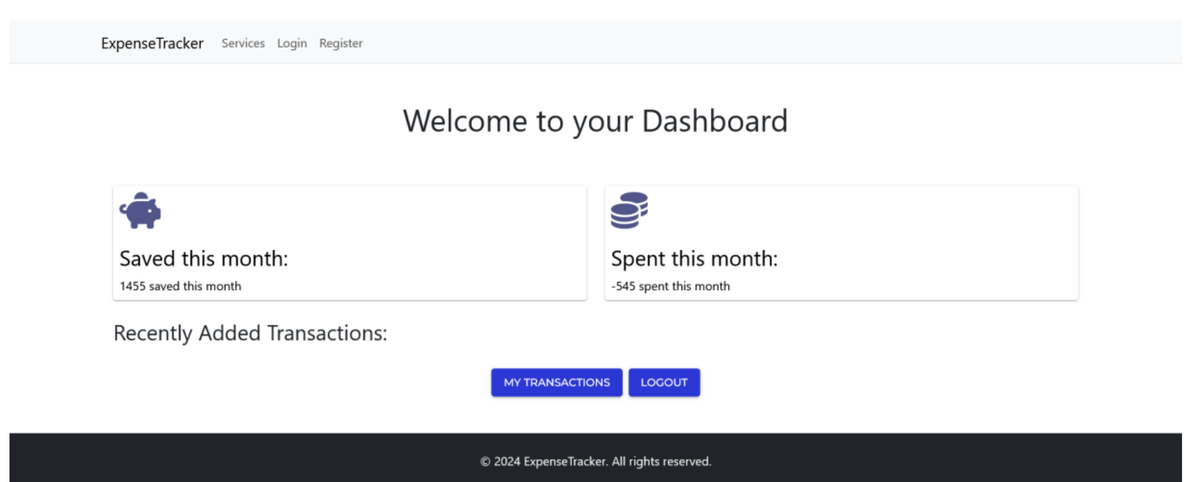
Na ovaj način se sve greške obrađuju na jednom mjestu bez ponavljanja istog bloka koda. Ovaj pristup prati Don't Repeat Yourself princip koji nalaže da se dupliciranje koda treba reducirati ponajviše radi lakšeg održavanja aplikacije i kasnijih promjena [45]. U tom slučaju bilo koja promjena bi zahtijevala ručnu izmjenu na svim mjestima gdje se taj dio koda pojavljuje.

4.5. Korisničko sučelje

Ovo poglavlje sadrži opis izrade korisničkog web sučelja koristeći ReactJS biblioteku. Korisničko sučelje sastoji se od početne stranice (Homepage) kojoj svaki neprijavljeni korisnik može pristupiti. Preko navigacijske trake, korisnik može otići na stranice „Services" gdje su opisane usluge koje nudi ova aplikacija, „Login" ako je već postojeći korisnik i „Register" ako želi kreirati račun.

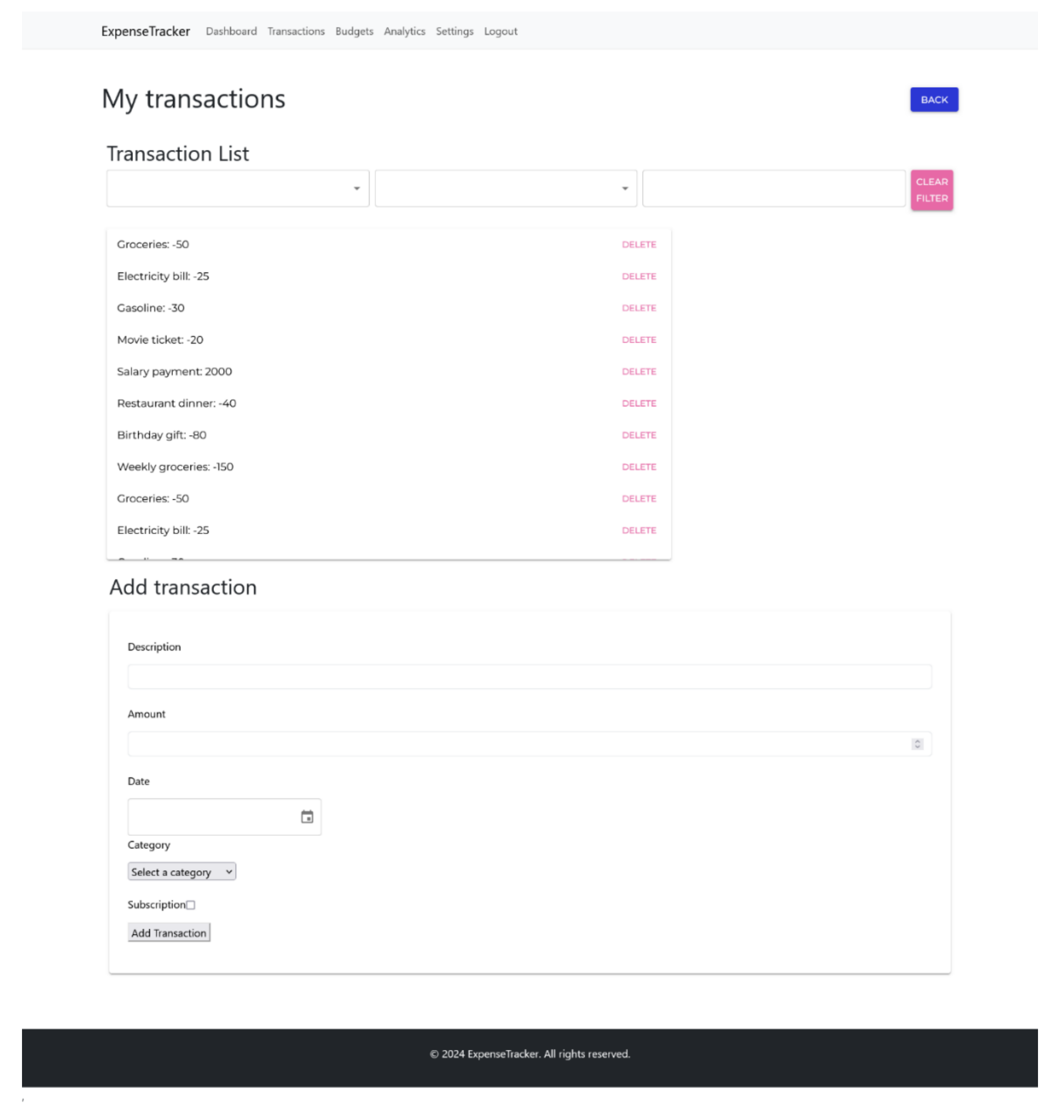
Nakon kreiranja računa, korisnik ima pristup sljedećim komponentama:

1) User dashboard (Slika 28): na ovoj stranici korisnik ima kratak uvidu u svoju aktivnost na aplikaciji. Na stranici se nalaze dvije kartice, jedna s ušteđenim iznosom u tekućem mjesecu, druga s potrošenim. Ispod na lijevoj strani je popis nedavno dodanih transakcija, a na desnoj pretplate koje ubrzo istječu i postoji vjerojatnost da se automatski naplate od strane trgovca.



Slika 27 User dashboard

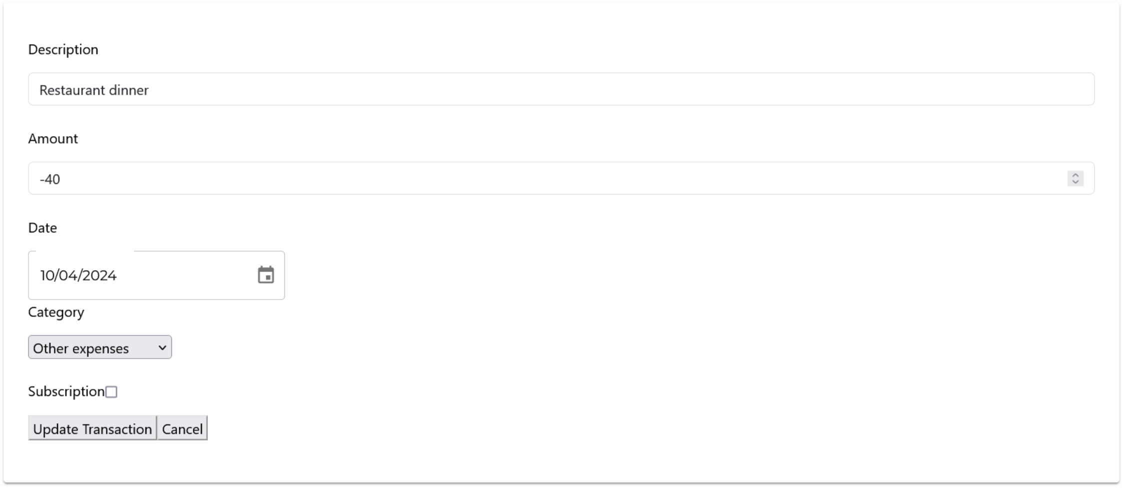
2) Transaction page (Slika 29): klikom na gumb „My transactions“ na user dashboard-u ili klikom na „Transactions“ na navigacijskoj traci, korisnik dolazi na stranicu gdje može vidjeti i filtirati svoje transakcije, uređivati ili brisati već dodane i dodati nove.



Slika 28 Transactions page

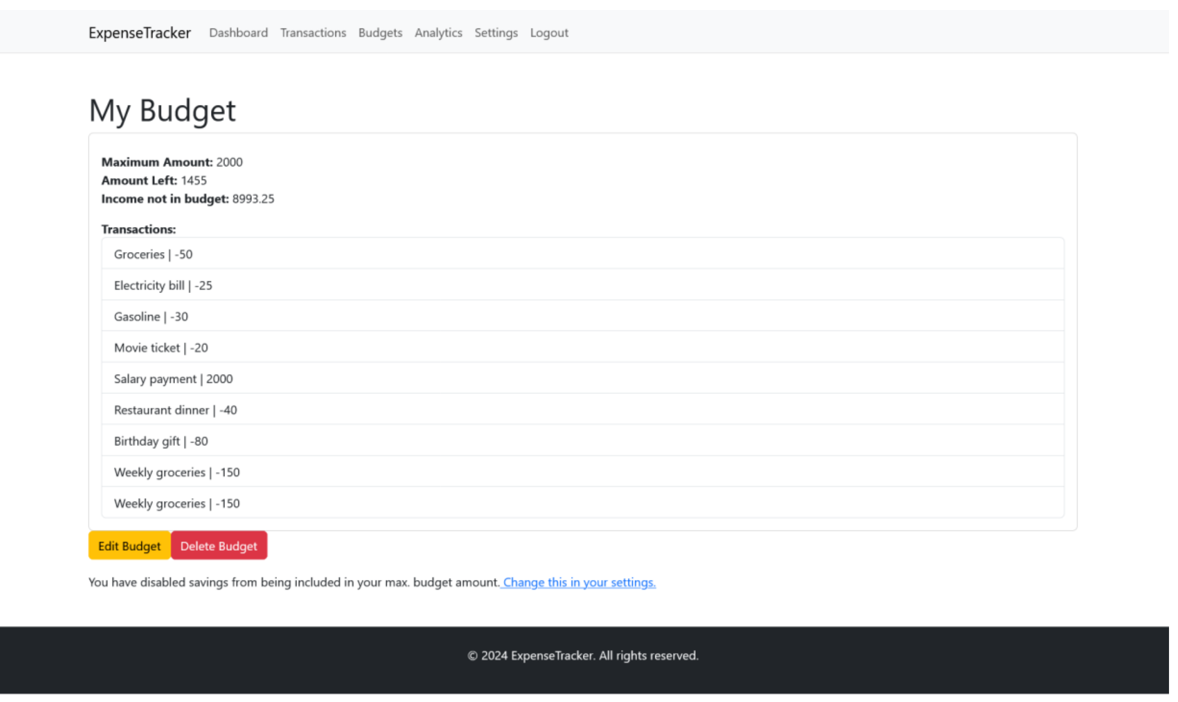
Klikom na transakciju, korisnik može vidjeti detalje transakcije (Slika 30). Klikom na gumb „Edit“ otvara se obrazac za uređivanje transakcije koji je vidljiv na Slici 31.

Slika 29 Detalji transakcije



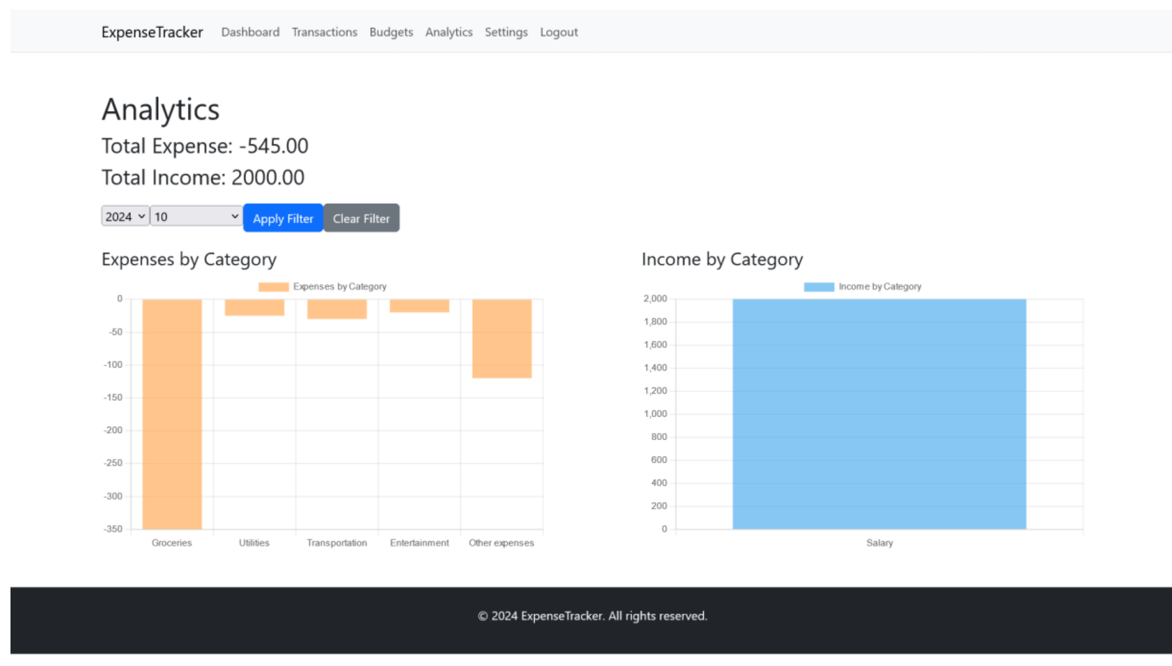
Slika 30 Edit transaction obrazac

3) Budget page (Slika 32): na ovoj stranici korisnik ima pregled svojeg trenutno aktivnog budžeta. Budžet se generira automatski prema vrijednosti „Monthly income“ i svakom dodanom transakcijom taj iznos se smanjuje ili povećava ovisno o tome je li transakcija trošak ili prihod.



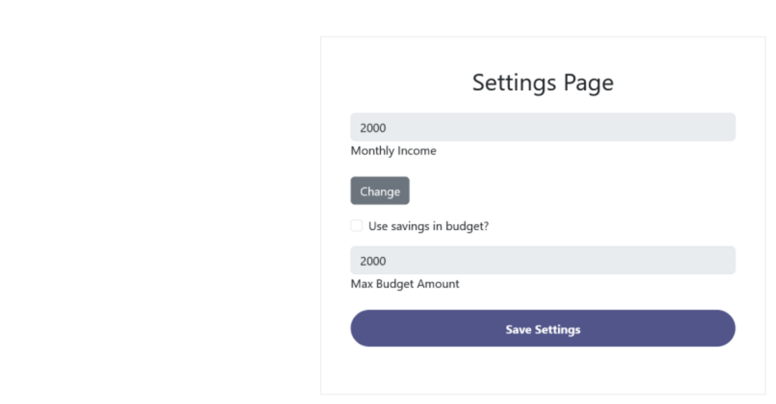
Slika 31 Budget page

4) Analytics page (Slika 33): na stranici Analytics korisnik ima pregled svojeg trošenja po kategorijama koje može dalje filtrirati po mjesecu ili godini.



Slika 32 Analytics page, prihodi filtrirani po tekućem mjesecu

5) Settings page (Slika 34): na ovoj stranici korisnik može promijeniti „Monthly income" iznos i dozvoliti aplikaciji da promijeni maksimalni iznos budžeta tako da koristi ušteđevinu. Ovo je samo moguće ako ušteđeno nije 0.



Slika 33 Settings page

4.5.1. Razvojni okviri

U ovom projektu korišten je JSX (JavaScript XML) što je zapravo proširenje Javascript sintakse koji omogućuje korištenje HTML-a unutar Javascript klasa [46]. Korišteni su i MaterialUI za dizajn korisničkog sučelja, days.js biblioteka za upravljanje datumima i fontawesome za upravljanje fotografijama. Za upravljanje CSS-om nije korišten razvojni okvir.

4.5.2. Povezivanje klijentske i serverske strane

Kada su i serverska i klijentska strana gotove, potrebno ih je povezati, tj. predati klijentskom sučelju putanje kako bi moglo upravljati podacima i komunicirati s bazom preko backend-a. React nudi biblioteke za lakše upravljanje HTTP zahtjevima poput axios-a, no u ovom projektu korišten je izvorni fetch koji nudi Javascript.

Povezivanje klijentskog i serverskoj dijela aplikacije zahtijeva dodatnu pažnju zbog provođenja sigurnosnih protokola kako bi se zaštitili podaci. Sigurnosni protokoli sprječavaju pozivanje putanja iz klijentskog dijela aplikacije radi toga što dolaze iz različitih izvora. To se naziva i CORS ili Cross-Origin Resource Sharing. CORS je sigurnosni mehanizam ugrađen u preglednike koji dozvoljava kontrolirani pristup podacima koji se nalaze izvan zadane domene [47]. Implementiran je kako bi se spriječili Cross-site scripting (XSS) i Cross-site Request Forgery (CSRF) napadi. XSS napadi su napadi u kojima se maliciozne skripte "ubrizgaju" u pouzdane web stranice. Ako se takva skripta izvrši, ona može onda pristupiti svim podacima koje je preglednik dobio od korisnika, što može uključivati i osjetljive podatke poput lozinki ili čak brojeva bankovnih kartica. Budući da se u ovom projektu koristi JWT ili JSON Web token koji se šalju natrag serverskom dijelu aplikacije za autentikaciju korisnika, ovakvi napadi bi mogli ugroziti sigurnost aplikacije. Stoga je na serverskoj strani implementirana kontrola izvora zahtjeva [48]. To se postiže na način da se iznad kontrolera ili metoda kontrolera postavi izvor tj. odakle dolaze zahtjevi, što nam omogućuje anotacija @CrossOrigin koju je moguće vidjeti na Slici 35. Na taj način serveru govorimo da vjerujemo toj putanji i da nju smije propustiti.

```
15 @RestController
16 @CrossOrigin(origins = "http://localhost:3000")
17 @RequestMapping("/users")
18 public class UserController {
```

Slika 34 Prikaz anotacija iznad UserController klase

Kao što je objašnjeno u dijelu omatanja odgovora, na serverskoj strani se također obavlja i validacija tokena. Iz JSON Web tokena izvuče se username korisnika i pregledava se postoji li taj korisnik u bazi podataka.

Encoded PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwiaWF0IjoxNzI2MDY4NDMyLCJleHAiOjE3MjYwNzIwMzJ9.ET98dVIwi76GSZrioE53z1dhnx6jNHZDQuCJbIbSEGY

Slika 35 Prikaz enkriptiranog JWT-a[49]

JWT se sastoji od tri dijela (Slika 36): header-a ili zaglavlja (crveno), payload-a ili tijela (ljubičasto) i potpisa (plavo). Da bi token bio ispravno dekodiran moramo predati i tajni ključ.

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256"
}
```

PAYLOAD: DATA

```
{
  "sub": "user",
  "iat": 1726068432,
  "exp": 1726072032
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  f30a43aa37634539f2b59
) ☑ secret base64 encoded
```

Slika 36 Prikaz dekriptiranog tokena[49]

Na Slici 37 vidimo kako se token dekriptira. Ono što je serveru bitno iz tokena je sub dio payload-a. Sub sadrži informaciju o korisniku, što je u ovom slučaju zapravo korisničko ime. Dekodiranjem tokena server dobiva korisničko ime i preko njega validira korisnika. Ako se korisničko ime ne nalazi u bazi podataka, token nije ispravan. Osim toga, token sadrži i podatak o isteku. U aplikaciji je postavljeno da token istječe nakon sat vremena, što se također može izvući iz tijela tokena.

U klijentskom dijelu aplikacije, prilikom prijave korisnika kao povratna informacija dobiva se token koji se sprema u localStorage što je zapravo značajka React-a koja omogućava developerima da spremaju i dohvaćaju podatke lokalno unutar web preglednika korisnika.

```
29      fetch('http://localhost:8085/auth/login', {
30          method: 'POST',
31          body: JSON.stringify(payload),
32          headers: {
33              'Content-Type': 'application/json'
34          }
35      })
36      .then(response => response.json())
37      .then(json => {
38          console.log(json);
39          if (json?.token) {
40              localStorage.setItem('authToken', json.token);
41              dispatch(setter(json.token));
42              navigate('/dashboard');
43          } else {
44              setError('Login failed.');
45          }
46      })
```

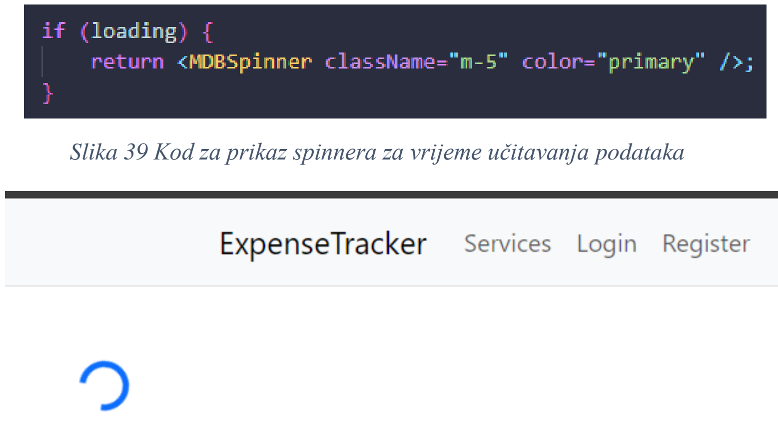
Slika 37 Poziv endpoint-a i spremanje tokena u localStorage

Korištenjem fetch-a šaljemo asinkrone HTTP zahtjeve serveru [50]. Ovakav način komunikacije omogućava da klijent ne mora čekati odgovor servera kako bi nastavio dalje sa izvršavanjem. Tako se poboljšava korisničko iskustvo jer sučelje ostaje responzivno dok se podaci dohvaćaju ili šalju. Uz to, na Slici 38 vidimo i kako upravljamo greškama. Ako odgovor od servera nije onakav kakav očekujemo, što je u ovom slučaju token, korisniku vraćamo povratnu informaciju i postavljamo grešku, kao što se i vidi na Slici 39.

Slika 38 Prikaz greške na sučelju

4.5.3. Omatanje odgovora

Nakon poziva endpoint-a, backend šalje odgovor natrag. Ovisno o dobivenom odgovoru korisnik vidi različite stvari na ekranu. Dok aplikacija učitava podatke, na ekranu se prikazuje spinner (MDBSpinner) kao što se to vidi na Slici 41, a kod koji omogućava prikaz spinnera je vidljiv na Slici 40.



Slika 39 Kod za prikaz spinnera za vrijeme učitavanja podataka

Slika 40 Prikaz spinner-a na sučelju

Ako postoji greška prilikom poziva backenda, korisniku se prikazuje poruka s detaljima o grešci. Da se dogodilo da backend nije bio aktivan, a klijent se pokušao prijaviti u svoj račun prikazala bi se poruka „Login failed“ na zaslonu (Slika 39).

4.6. Automatizacija i raspoređivanje poslovne logike

Kako bi ova aplikacija mogla raditi bez konstantnog nazora čovjeka i bez izvršavanja gomile koda sa svakim novim korisnikom ili svakom dodanom transakcijom, u aplikaciji su uvedeni triggeri na bazi i „schedulirana“ klasa na backend-u.

4.6.1. Implementacija trigger-a

Triggeri na bazi okidaju se automatski s registracijom korisnika, promjenom postavki, unosom nove transakcije ili promjenom postojeće.

Na unos novog sloga u tablicu USERS, kreira se slog u tablicama USERS_HISTORY, USER_SETTINGS i BUDGETS. USERS_HISTORY je kronološka tablica koja je kreirana u svrhu vođenja kronologije tj. povijesti korisnika. Na ovaj način možemo voditi računa npr. kad je korisnik zadnji put promijenio lozinku ili je li ta lozinka ista trenutnoj ako korisnik zaboravi lozinku i želi ju promijeniti. USER_SETTINGS je pomoćna tablica u kojoj se nalaze postavke korisničkog računa. Kreirana je u svrhu olakšavanja vođenja dodatnih

podataka o korisniku i bilježenja financijskih podataka kao npr. „total_savings“ kolona u koju se sa svakom transakcijom zabilježi koliko je ušteđeno.

Osim toga ovih trigger-a, na tablici TRANSACTIONS postavljena su tri triggera koji se okidaju sa svakim unosom nove transakcije. Ažurira se slog u USER_SETTINGS gdje se ažurira iznos „total_savings“ kolone za korisnika, ažurira se preostali iznos budžeta u tablici BUDGETS, ali samo ako je transakcija napravljena u tekućem mjesecu i unosi se slog u tablicu SUBSCRIPTIONS samo ako je transakcija označena kao pretplata. Na ovaj način imamo lako dostupne podatke koje nije potrebno izračunavati kroz backend što štedi i vrijeme i novac.

4.6.2. @Scheduled metoda

Korisnik prilikom registracije unosi dan u mjesecu kad mu odgovara da mu se resetira obračun. Na taj dan korisnik dobiva novi budžet i puni iznos na trošenje. Sve nepotrošeno prelazi u ušteđevinu. Kako netko svaki dan ne bi morao puštati neku obradu koja bi provjeravala treba li korisniku resetirati iznos budžeta prema određenom reset danu, uvedena je zakazana metoda ili metoda sa @Scheduled anotacijom [51].

U aplikaciji je metoda implementirana koristeći cron izraz. Cron je program dostupan na Unixu sustavima [52]. Korištenjem cron-a korisnici mogu zakazati pokretanje određenog zadatka po rasporedu koji oni sami odrede. Na taj način zadaci koji bi inače trebali ljudsku intervenciju se odrade automatski. Tako je u ovoj aplikaciji napravljena @Scheduled metoda koja se automatski pokreće svaki dan u 5 ujutro i provjerava kome treba resetirati budžet.

5. Rasprava

Ova web aplikacija predstavlja inicijalnu verziju aplikacije za praćenje vlastitih troškova i upravljanje osobnim financijama. Razvijena je korištenjem tehnologija kao što su Spring Boot, ReactJS i MySQL baza podataka koje omogućuju jednostavnu implementaciju i održavanje sustava. Ove tehnologije pružaju snažnu osnovu za izgradnju skalabilnih i responzivnih full stack aplikacija. Spring Boot omogućuje brzo postavljanje i razvoj backend-a i jednostavnu integraciju s bazama podataka putem pluginova Spring JPA i Hibernate. Na frontend-u, ReactJS omogućava razvoj interaktivnih korisničkih sučelja koja se brzo ažuriraju bez potrebe za ponovnim učitavanjem stranica. Za pohranu podataka, MySQL je idealno rješenje jer podržava obavljanje složenih upita i transakcija.

Trenutna verzija aplikacije postoji samo lokalno, no kako bi postala javno dostupna korisnicima, potrebno ju je postaviti u produkciju. To se može postići na način da se aplikacija kontejnerizira. Kontejnerizacija podrazumijeva pakiranje aplikacije zajedno s njenim kodom, bibliotekama operativnog sustava i svim potrebnim ovisnostima u jedan lagani, samostalni izvršni paket, poznat kao kontejner [53]. Ovaj kontejner može se pokrenuti na bilo kojoj infrastrukturi, bez obzira na specifične karakteristike operativnog sustava ili okruženja.

Kontejneri omogućuju pouzdano pokretanje aplikacije u različitim okruženjima, a to je osobito korisno kada se koriste cloud platforme poput AWS, Google Cloud ili Microsoft Azure. Upotrebom tehnologija poput Docker-a, platforme koja omogućuje izgradnju, testiranje i razvoj aplikacija u izoliranim okruženjima, omogućava se brzo i učinkovito skaliranje te jednostavno upravljanje aplikacijom.

Uz trenutno postojeće rješenje koje predstavlja osnovnu, funkcionalnu verziju aplikacije moguće je aplikaciju unaprijediti na način da se poveže s mobilnim novčanicima kao što su to npr. Google Wallet ili Apple Pay (aplikacije za mobilno plaćanje koje omogućuju korisniku plaćanje mobitelom bez korištenja fizičkih kartica[54], [55]) gdje bi se korisniku sa svakom transakcijom plaćenom mobitelom unijela ta ista transakcija u aplikaciju. Ujedno, kako su danas sve više u upotrebi i aplikacije mobilnog bankarstva, povezivanje ove aplikacije s m-bankingom olakšalo bi praćenje troškova. U aplikacijama mobilnog bankarstva korisnici mogu pregledavati sve svoje transakcije (prihode i troškove), a nakon povezivanja s aplikacijom za praćenje troškova transakcije bi se automatski unosile nakon izvršenja, bez potrebe za ručnim unosom od strane korisnika.

Da bi se to moglo ostvariti potrebno bi bilo dodatno pojačati sigurnosnu komponentu koja je trenutno implementirana jer su podaci o plaćanju i o karticama osjetljivi podaci koje treba posebno pohranjivati kako ne bi došlo do curenja podataka i nanošenja štete korisniku. Uz ovo se vežu i zakonske regulative koje nalažu na koji način se osobni te osjetljivi kartični podaci moraju pohranjivati. Sve organizacije koje rade s kartičnim podacima, moraju poštivati i raditi u skladu s PCI DSS regulativom. PCI DSS (Payment Card Industry Data Security Standard) je standard u kartičnom poslovanju i stvoren je kako bi se potaknula i poboljšala zaštita podataka o plaćanju[56]. Osnovan je od strane kartičnih kuća i sastoji se od dvanaest uvjeta koji se trebaju ispuniti. Najbitniji uvjeti u sklopu unaprjeđenja ove aplikacije bi bili točka 3 i točka 4. Točka 3 kaže da se spremljeni podaci o računu korisnika moraju zaštititi, a točka 4 kaže da se podaci o vlasniku kartice moraju zaštiti snažnom

kriptografijom prilikom prenošenja podataka tj. prilikom komunikacije između odvojenih resursa (u ovom slučaju aplikacija za praćenje troškova i npr. mobilno bankarstvo)[56].

6. Zaključak

Razvijena aplikacija „ExpenseTracker" pruža korisnicima učinkovit alat za praćenje njihovih troškova i prihoda. Korištenjem Spring Boot-a za razvoj backend-a, ReactJS-a za razvoj korisničkog sučelja te MySQL baze podataka omogućila se stabilna i fleksibilna arhitektura aplikacije koja se dalje može proširivati i prilagođavati budućim potrebama korisnika.

Integracijom sigurnosnih mehanizama poput JWT autentifikacije i Spring Security implementacije osigurala se zaštita korisničkih podataka, što je ključno u modernim web aplikacijama. Nadalje, upravljanje podacima se olakšalo i pojednostavnilo korištenjem tehnika poput DI i ORM alata (Hibernate, MyBatis).

Iako ova aplikacija ispunjava svoje osnovne funkcionalnosti, postoji prostor za daljnji razvoj, poput implementacije naprednijih analitičkih alata ili integracije s financijskim servisima u stvarnom vremenu. Takva proširenja mogla bi dodatno unaprijediti korisničko iskustvo i omogućiti korisnicima precizniju kontrolu nad svojim financijama.

Sve u svemu, ova aplikacija dobar je primjer fullstack aplikacije koja rješava stvaran problem s kojim se mnogi korisnici svakodnevno susreću.

7. Literatura

[1] K. Haan, „People Are Twice As Likely To Spend More Money When Using Card Than Cash In 2024“, Forbes Advisor. Pristupljeno: 28. kolovoz 2024. [Na internetu]. Dostupno na: https://www.forbes.com/advisor/business/software/people-twice-likely-spend-using-card-than-cash/

[2] S. Walden, „What Is A Neobank?“, Forbes Advisor. Pristupljeno: 30. srpanj 2024. [Na internetu]. Dostupno na: https://www.forbes.com/advisor/banking/what-is-a-neobank/

[3] „Revolut | All-in-one Finance App for your Money | Revolut United Kingdom“, Revolut. Pristupljeno: 30. srpanj 2024. [Na internetu]. Dostupno na: https://www.revolut.com/

[4] „What Is A Technology Stack? Tech Stacks Explained“, MongoDB. Pristupljeno: 28. kolovoz 2024. [Na internetu]. Dostupno na: https://www.mongodb.com/resources/basics/technology-stack

[5] „What Is Full Stack Development? | A Complete Guide“, MongoDB. Pristupljeno: 12. lipanj 2024. [Na internetu]. Dostupno na: https://www.mongodb.com/resources/basics/full-stack-development

[6] „What is a Programming Library? A Beginner’s Guide“. Pristupljeno: 24. rujan 2024. [Na internetu]. Dostupno na: https://careerfoundry.com/en/blog/web-development/programming-library-guide/

[7] R. E. Johnson, „Frameworks = (components + patterns)“, *Commun. ACM*, sv. 40, izd. 10, str. 39–42, lis. 1997, doi: 10.1145/262793.262799.

[8] „18. Using the @SpringBootApplication Annotation“. Pristupljeno: 13. kolovoz 2024. [Na internetu]. Dostupno na: https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-using-springbootapplication-annotation.html

[9] „EnableAutoConfiguration (Spring Boot 3.3.2 API)“. Pristupljeno: 13. kolovoz 2024. [Na internetu]. Dostupno na: https://docs.spring.io/spring-boot/api/java/org/springframework/boot/autoconfigure/EnableAutoConfiguration.html

[10] „What Is a Spring Bean? | Baeldung“. Pristupljeno: 22. listopad 2024. [Na internetu]. Dostupno na: https://www.baeldung.com/spring-bean

[11] „ComponentScan (Spring Framework 6.1.11 API)“. Pristupljeno: 13. kolovoz 2024. [Na internetu]. Dostupno na: https://docs.spring.io/spring-framework/docs/6.1.11/javadoc-api/org/springframework/context/annotation/ComponentScan.html

[12] „5. The IoC container“. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html

[13] „Maven – Welcome to Apache Maven“. Pristupljeno: 18. kolovoz 2024. [Na internetu]. Dostupno na: https://maven.apache.org/

[14] „Maven – Introduction to the POM“. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

[15] „Database Management Systems (DBMS) Comparison: MySQL, Postgr“, AltexSoft. Pristupljeno: 17. lipanj 2024. [Na internetu]. Dostupno na: https://www.altexsoft.com/blog/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/

[16] „JSON“. Pristupljeno: 22. listopad 2024. [Na internetu]. Dostupno na: https://www.json.org/json-en.html

[17] „What Is NoSQL? NoSQL Databases Explained“, MongoDB. Pristupljeno: 22. listopad 2024. [Na internetu]. Dostupno na: https://www.mongodb.com/resources/basics/databases/nosql-explained
[18] „What is the Document Object Model?“ Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://www.w3.org/TR/WD-DOM/introduction.html
[19] „React JS ReactDOM“, GeeksforGeeks. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://www.geeksforgeeks.org/reactjs-reactdom/
[20] „Virtual DOM and Internals – React“. Pristupljeno: 22. listopad 2024. [Na internetu]. Dostupno na: https://legacy.reactjs.org/docs/faq-internals.html
[21] „Spring Boot Architecture - javatpoint“. Pristupljeno: 17. kolovoz 2024. [Na internetu]. Dostupno na: https://www.javatpoint.com/spring-boot-architecture
[22] „mybatis – MyBatis 3 | Introduction“. Pristupljeno: 28. kolovoz 2024. [Na internetu]. Dostupno na: https://mybatis.org/mybatis-3/
[23] „Project Lombok“. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://projectlombok.org/
[24] „Spring Boot - Difference Between CrudRepository and JpaRepository“, GeeksforGeeks. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://www.geeksforgeeks.org/spring-boot-difference-between-crudrepository-and-jparepository/
[25] „Pro EJB 3 - Java Persistence API (2006).pdf“. Pristupljeno: 16. rujan 2024. [Na internetu]. Dostupno na: https://theswissbay.ch/pdf/Gentoomen%20Library/Programming/Java/Pro%20EJB%203%20-%20Java%20Persistence%20API%20(2006).pdf
[26] „What is data persistence?“, FutureLearn. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://www.futurelearn.com/info/blog
[27] „Hibernate - @GeneratedValue Annotation in JPA“, GeeksforGeeks. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://www.geeksforgeeks.org/hibernate-generatedvalue-annotation-in-jpa/
[28] „Hibernate Tutorial For Beginners | DigitalOcean“. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://www.digitalocean.com/community/tutorials/hibernate-tutorial-for-beginners
[29] „The Spring @Controller and @RestController Annotations | Baeldung“. Pristupljeno: 03. rujan 2024. [Na internetu]. Dostupno na: https://www.baeldung.com/spring-controller-vs-restcontroller
[30] „What are HTTP Requests? | Cloud4Y“. Pristupljeno: 23. listopad 2024. [Na internetu]. Dostupno na: https://www.cloud4y.ru/en/blog/what-are-http-requests/
[31] „HTTP request methods - HTTP | MDN“. Pristupljeno: 01. rujan 2024. [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
[32] „@RequestBody :: Spring Framework“. Pristupljeno: 03. rujan 2024. [Na internetu]. Dostupno na: https://docs.spring.io/spring-framework/reference/web/webflux/controller/ann-methods/requestbody.html
[33] „Core Spring Concept - Dependency Injection (DI)“. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://www.linkedin.com/pulse/core-spring-concepts-ognyana-stefanova-bokwf
[34] „Is Java Reflection Bad Practice? | Baeldung“. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://www.baeldung.com/java-reflection-benefits-drawbacks
[35] „Why Is Field Injection Not Recommended? | Baeldung“. Pristupljeno: 03. rujan 2024. [Na internetu]. Dostupno na: https://www.baeldung.com/java-spring-field-injection-cons

[36] Hjwasim, „Where can we initialize the final instance variables?“, Medium. Pristupljeno: 07. rujan 2024. [Na internetu]. Dostupno na: https://medium.com/@hjwasim/where-can-we-initialize-the-final-instance-variables-2192f2753b11
[37] baeldung, „Constructor Dependency Injection in Spring | Baeldung“. Pristupljeno: 03. rujan 2024. [Na internetu]. Dostupno na: https://www.baeldung.com/constructor-injection-in-spring
[38] H. Shukla, „#SpringSecurity Part 1 : Authentication v/s Authorization“, Medium. Pristupljeno: 03. rujan 2024. [Na internetu]. Dostupno na: https://medium.com/@greekykhs/springsecurity-part-1-authentication-v-s-authorization-92445548dfbe
[39] „Authorization - HTTP | MDN“. Pristupljeno: 03. studeni 2024. [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization
[40] „User agent - MDN Web Docs Glossary: Definitions of Web-related terms | MDN“. Pristupljeno: 03. studeni 2024. [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Glossary/User_agent
[41] „What is Authorization? - Examples and definition“, Auth0. Pristupljeno: 03. rujan 2024. [Na internetu]. Dostupno na: https://auth0.com/intro-to-iam/what-is-authorization
[42] „Lambda Expressions (The Java™ Tutorials > Learning the Java Language > Classes and Objects)“. Pristupljeno: 24. rujan 2024. [Na internetu]. Dostupno na: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
[43] „Controller Advice :: Spring Framework“. Pristupljeno: 08. rujan 2024. [Na internetu]. Dostupno na: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-advice.html
[44] „RuntimeException (Java Platform SE 8)“. Pristupljeno: 09. rujan 2024. [Na internetu]. Dostupno na: https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html
[45] „What is DRY? Hint: It makes for great code - dbt Labs“. Pristupljeno: 09. rujan 2024. [Na internetu]. Dostupno na: https://docs.getdbt.com/terms/dry
[46] Y. Kods, „Is JS and JSX the Same?“, Medium. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://medium.com/@ykods/is-js-and-jsx-the-same-97e4df644609
[47] „Cross-Origin Resource Sharing (CORS) - HTTP | MDN“. Pristupljeno: 01. rujan 2024. [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[48] A. Obregon, „Spring Framework — Unraveling the @Service Annotation“, Medium. Pristupljeno: 30. kolovoz 2024. [Na internetu]. Dostupno na: https://medium.com/@AlexanderObregon/spring-framework-unraveling-the-service-annotation-363f7d1e55e6
[49] „JSON Web Tokens - jwt.io“. Pristupljeno: 23. listopad 2024. [Na internetu]. Dostupno na: https://jwt.io/
[50] „How to Use Fetch with async/await“, Dmitri Pavlutin Blog. Pristupljeno: 14. rujan 2024. [Na internetu]. Dostupno na: https://dmitripavlutin.com/javascript-fetch-async-await/
[51] „The @Scheduled Annotation in Spring | Baeldung“. Pristupljeno: 13. listopad 2024. [Na internetu]. Dostupno na: https://www.baeldung.com/spring-scheduled-tasks
[52] „A Guide To Cron Expressions | Baeldung“. Pristupljeno: 13. listopad 2024. [Na internetu]. Dostupno na: https://www.baeldung.com/cron-expressions

[53] „What Is Containerization? | IBM“. Pristupljeno: 10. studeni 2024. [Na internetu]. Dostupno na: https://www.ibm.com/topics/containerization
[54] „About Google Wallet - Google Wallet Help“. Pristupljeno: 13. listopad 2024. [Na internetu]. Dostupno na: https://support.google.com/wallet/answer/11951709?hl=en
[55] „Apple Pay“, Apple. Pristupljeno: 13. listopad 2024. [Na internetu]. Dostupno na: https://www.apple.com/apple-pay/
[56] „Payment Card Industry Data Security Standard“, 2024.

8. Popis slika

Slika 1 Principjelni prikaz Full stack razvoja[5] ... 3
Slika 2 Primjer upotrebe anotacije iz projekta autorice ... 5
Slika 3 Relacijske vs ne-relacijske baze podataka[15] ... 7
Slika 4 Grafički prikaz troslojne arhitekture aplikacija[21] ... 11
Slika 5 Konceptualni prikaz rješenja ... 12
Slika 6 Prikaz tablice Users na bazi ... 13
Slika 7 Prikaz tablice Budgets na bazi ... 13
Slika 8 Prikaz tablice Transactions na bazi ... 14
Slika 9 Prikaz tablice Subscriptions na bazi ... 14
Slika 10 Prikaz tablice Categories na bazi ... 14
Slika 11 Prikaz User_settings Budgets na bazi ... 15
Slika 12 Prikaz tablice User_settings_history na bazi ... 15
Slika 13 Prikaz tablice Users_history na bazi ... 16
Slika 14 Prikaz tablice Budgets_history na bazi ... 16
Slika 15 Primjer iz projekta autorice ... 17
Slika 17 Primjer iz projekta autorice (application.properties datoteka) ... 17
Slika 18 Prikaz UserRepository klase ... 18
Slika 19 Prikaz User klase ... 19
Slika 20 Objašnjenje HTTP zahtjeva[30] ... 21
Slika 21 Prikaz poziva endpoint-a iz Postmana ... 22
Slika 22 Prikaz enpoint-a za prijavu korisnika ... 22
Slika 23 Prikaz HTTP zahtjeva s Authorization header-om u React-u ... 24
Slika 24 Prikaz prava pristupa ... 25
Slika 25 Prikaz @ControllerAdvice klase ... 26
Slika 26 Prikaz endpoint-a za dodavanje transakcije ... 26

Slika 27 Prikaz ponavljajućeg koda koji se izvukao u zasebne metode 27
Slika 28 User dashboard 28
Slika 29 Transactions page 29
Slika 30 Detalji transakcije 29
Slika 31 Edit transaction obrazac 30
Slika 32 Budget page 30
Slika 33 Analytics page, prihodi filtrirani po tekućem mjesecu 31
Slika 34 Settings page 31
Slika 35 Prikaz anotacija iznad UserController klase 32
Slika 36 Prikaz enkriptiranog JWT-a[49] 33
Slika 37 Prikaz dekriptiranog tokena[49] 33
Slika 38 Poziv endpoint-a i spremanje tokena u localStorage 34
Slika 39 Prikaz greške na sučelju 34
Slika 40 Kod za prikaz spinnera za vrijeme učitavanja podataka 35
Slika 41 Prikaz spinner-a na sučelju 35

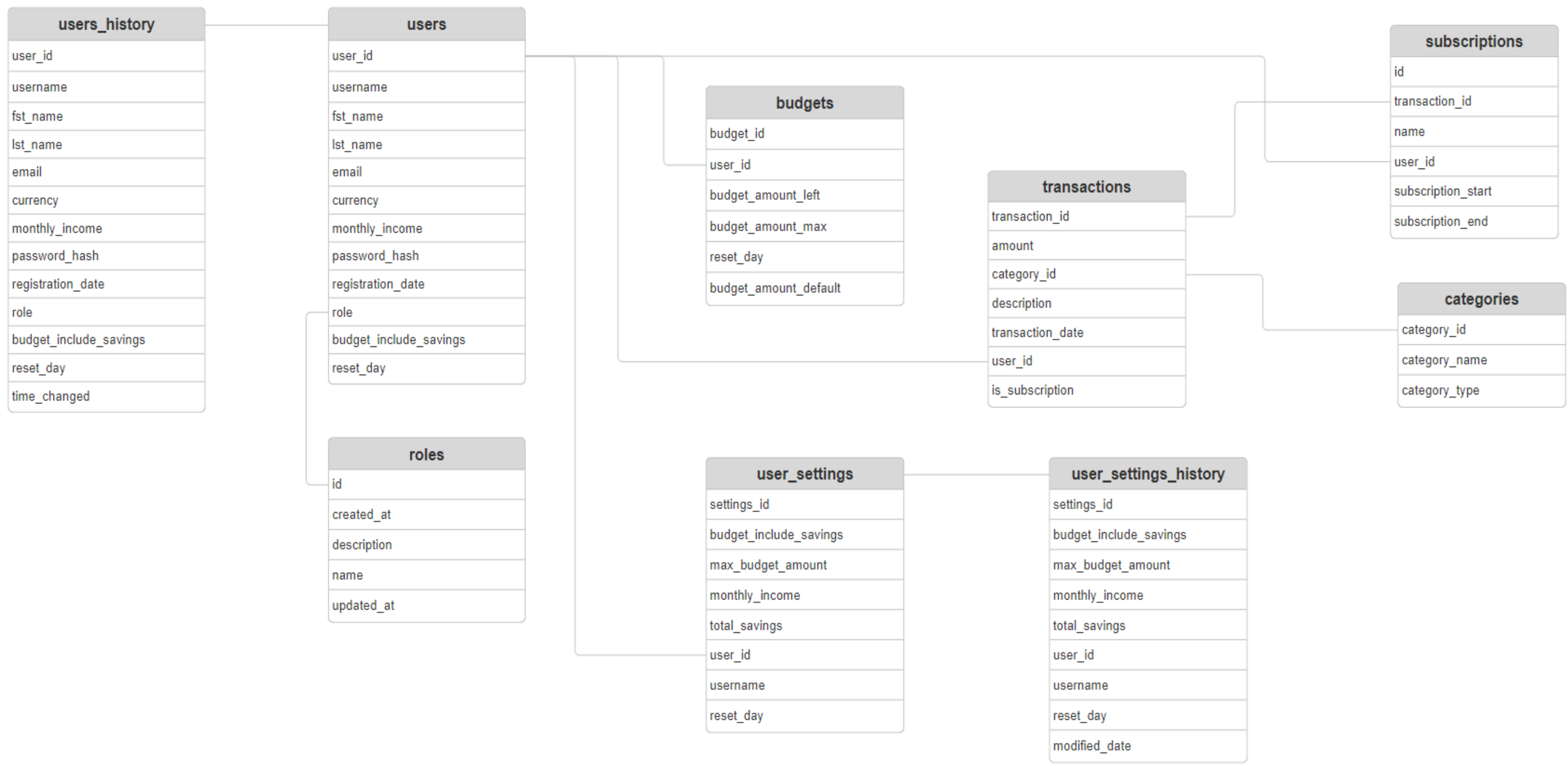
Fullstack development of a web application for tracking personal expenses

SUMMARY:

This paper presents and describes the development of the full-stack web application „ExpenseTracker." Spring Boot was used for the development of the server part of the application or backend, and ReactJS was used for the frontend or the user interface. The application offers a platform for monitoring personal expenses through a detailed overview of expenses and savings in a given period of time by defined categories. By entering each transaction, be it income or expense, the user can add a detailed description, the date of execution, and whether the transaction is part of a subscription. The advantages of this application are the display of expenses and the implementation of a monthly budget, which enabled the user to continuously monitor the amount allowed for spending. This application can be further developed and improved, mostly in such a way as to allow the user to connect to their mobile banking application, which would speed up the addition of transactions to the application.

KEYWORDS: Spring Boot, ReactJS, expenses, MySQL, full-stack, web application

Prilozi



Dodatak 1 ERD dijagram projekta