DIZAJN I IMPLEMENTACIJA CJELOVITOG RASPODIJELJENOG SUSTAVA ZA KONTROLU I VOĐENJE OSOBNIH FINANCIJA

Rajković, Eugen

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

DIZAJN I IMPLEMENTACIJA CJELOVITOG RASPODIJELJENOG SUSTAVA ZA KONTROLU I VOĐENJE OSOBNIH FINANCIJA

Eugen Rajković

Zagreb, veljača 2020.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, veljača 2020.

Predgovor

Zahvaljujem svom mentoru Aleksanderu Radovanu na ukazanom povjerenju, pomoći i savjetima prilikom izrade ovog završnog rada.

Također, želim zahvaliti svojim prijateljima koje sam upoznao na Visokom učilištu Algebra koji su učinili studiranje lakšim. Zahvaljujem svojoj obitelji jer su mi bili potpora tijekom cijelog školovanja. Najviše od svih želim zahvaliti svojoj zaručnici, čija je potpora i beskrajna motivacija bila oslonac pri pisanju ovoga rada.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Ovim završnim radom prikazani su dizajn i izrada raspodijeljenog sustava za kontrolu i vođenje osobnih financija, koji se temelji na klijent-poslužitelj arhitekturi. Pri razvoju sustava obuhvaćen je niz aktualnih tehnologija. Poslužitelj je implementiran u ASP.NET Core 3.0 tehnologiji. Na klijentskoj strani web aplikacija izrađena je u Vue.js, a mobilna u NativeScript-Vue programskom okviru. Komunikacija između klijenta i poslužitelja odvija se REST pozivima, koristeći HTTPS mrežni protokol. Za pristup podacima iz baze podataka upotrijebljen je Entity Framework Core, a za pohranu SQL Server baza podataka. Visual Studio 2019 i Visual Studio Code poslužili su kao razvojni alati. Cilj ovog rada je razvoj jednostavnog, pristupačnog i skalabilnog raspodijeljenog sustava za vođenje financija. Uz prikaz razvoja, u radu se želi opisati tehnologije koje su primijenjene za izradu sustava te njihove prednosti i ograničenja.

Ključne riječi: raspodijeljeni sustav, klijent-poslužitelj arhitektura, ASP.NET Core 3.0, Vue.js, NativeScript-Vue, Entity Framework Core

Summary

This final paper describes the design and development of a distributed system for managing personal finances, based on client-server architecture. A number of current technologies were included in the development of the system. The implementation of the server was performed in ASP.NET Core 3.0 technology. On the client side, a web application is created in the Vue.js framework, and a mobile application in the NativeScript-Vue framework. Client-server communication is secured by REST calls, using the HTTPS network protocol. Entity Framework Core was used to access the data from the database and SQL Server to store the data. Visual Studio 2019 and Visual Studio Code were used as development tools. The aim of this paper is to develop a simple, affordable and scalable distributed financial management system. In addition to presenting the development, the paper seeks to describe the technologies used to design the system and their advantages and limitations.

Key words: distributed system, client-server architecture, ASP.NET Core 3.0, Vue.js, NativeScript-vue, Entity Framework Core

Sadržaj

1. Uvod

Vođenje osobnih financija jedna je od neizbježnih životnih stavki većini osoba. Predstavlja osnovu financijske pismenosti, a znači mogućnost praćenja prihoda i rashoda. Bitno je za ostvarenje kratkoročnih i dugoročnih ciljeva te smanjenje potencijalnih nepredviđenih rizika. U prošlosti prihodi i troškovi često su se „stavljali na papir“. S razvojem tehnologije i većom informatičkom obrazovanošću mlađih generacija razvijali su se i načini praćenja troškova. U digitalnoj eri papir je zamijenjen najprije jednostavnim Excel tablicama koje su se prilagođavale vlastitim potrebama, a kasnije moćnijim računalnim i mobilnim alatima. Broj mogućnosti u praćenju osobnih financija postao je toliko veliki da ponekad djeluje zbunjujuće na korisnika. Zbog toga se javlja potreba smanjenja kognitivne i perceptivne „zagušenosti“ korisnika. Stoga je svrha ovog rada razvoj jednostavnog, pristupačnog i skalabilnog raspodijeljenog sustava za vođenje financija. Uz prikaz razvoja naglasak je stavljen i na recentne tehnologije koje su primijenjene za izradu sustava te njihove prednosti i ograničenja u odnosu na dosadašnja rješenja pri izradi softverskih proizvoda. Izrađeni sustav temelji se na klijent-poslužitelj arhitekturi pri čemu je poslužitelj implementiran u ASP.NET Core 3.0 tehnologiji, a klijentsku stranu čine web i mobilna aplikacija. Web aplikacija izrađena je u Vue.js, a mobilna u NativeScript-Vue programskom okviru koji je dizajniran za rad na svim mobilnim platformama. Kao razvojni alat korišten je Visual Studio 2019.

Uvod u tematiku rada obrađen je u drugom poglavlju, a u trećem poglavlju opisana je arhitektura sustava kroz sve njezine elemente – bazu podataka, sloj jezgre, infrastrukturni sloj, servisni sloj, aplikacijsko programsko sučelje (engl. *Application programming interface,* skraćeno API) te web i mobilnu aplikaciju kao klijente. Zaštita aplikacije osigurana je primjenom *Hypertext Transfer Protocol Secure* (skraćeno HTTPS) mrežnog protokola što je opisano u četvrtom poglavlju, gdje je razrađena i autorizacija i autentikacija korisnika. Također, u trećem i četvrtom poglavlju ukratko su objašnjene i tehnologije koje su korištene za razvoj raspodijeljenog sustava, a spomenute su u uvodnom dijelu. U petom poglavlju prikazano je korištenje aplikacije sa svojim osnovnim značajkama poput unošenja prihoda i troškova te grafički pregled istih za različite vremenske intervale. Predstavljen je i „sustav omotnica“ koji omogućuje praćenje ušteda i postavljanje ograničenja na određene potrošačke kategorije. U šestom poglavlju učinjena je analiza funkcionalnosti aplikacije kroz usporedbu s postojećim rješenjima i moguća proširenja aplikacije.

2. Osobne financije

Rijetko koji dio ekonomije je toliko primjenjiv i prisutan u svakodnevnom životu koliko su osobne financije. Proces i strategije učinkovitog upravljanja novcem i osobnim financijama u fokusu su ne samo ekonomista, već i šire javnosti. Takvo stanje nije čudno jer je novac univerzalna roba i čini sastavni dio života svakog pojedinca i cijelog gospodarstva [1]. Svaki pojedinac odlučuje o štednji, investicijama i potrošnji te tako utječe na vlastito financijsko stanje, ali i na financijsko stanje cijelog društva.

Osobne financije označavaju dio financija koji proučava načine korištenja osobnih i obiteljskih resursa s ciljem postizanja financijskog uspjeha. Razvijanje financijskog znanja i financijsko opismenjavanje neki su od načina koje pojedinci mogu naučiti kako bi se zaštitili od financijskih rizika, izbjegli porezne probleme, naučili kako štedjeti svoj novac i u što ga investirati.

Budžet je financijski plan pomoću kojeg pojedinac/kućanstvo alocira svoj budući dohodak na buduće rashode (troškove), štednju (investicije) i otplatu dugova (Cvrlje, 2010). Budžet je osnovni mehanizam osiguranja koji pojedinac može koristiti s ciljem iskorištavanja novca na najpotrebniji način, a uzimajući u obzir trenutnu financijsku situaciju. Čak i ako se osoba nalazi u povoljnoj financijskoj situaciji te ostvaruje visoke prihode, proces kreiranja budžeta ipak je vrlo koristan jer može dovesti do spoznaje da se na određene stvari troši značajno više no što je bilo predviđeno.

Upravljanje osobnim financijama važno je jer omogućuje poboljšanje životnog standarda. Dokazano je da donošenje dobrih financijskih odluka može dovesti do smanjenja siromaštva i dugovanja, a s druge strane povećati udio uštođenog i uloženog novca [2]. Dakle, financijsko planiranje odražava se na sve aspekte privatnog i poslovnog života te doprinosi smanjenju financijskog stresa i životnoj sigurnosti.

2.1. Postojeći alati na tržištu

Na tržištu trenutno postoji mnoštvo aplikacija i alata koji omogućuju praćenje i analizu financija. Na temelju prikupljenih informacija s korisničkog računa daju uvid u korisnikove navike, pomažu u donošenju financijskih odluka i upozoravaju na previđene izdatke. Aplikacije su najčešće dostupne na webu i na mobilnim uređajima. U općoj populaciji njihova uporaba, kao i općenito financijskih aplikacija, sve više raste. Istraživanja pokazuju da gotovo 20% Amerikanaca koristi neku od aplikacija za praćenje ili upravljanje novcem [3]. U nastavku je dan kratki pregled nekih od globalno najpopularnijih aplikacija za praćenje troškova: Mint, You Need A Budget, Wally i Toshl.

2.1.1. Mint

Mint je "zlatni standard" među aplikacijama za praćenje osobnih financija, a proizvod je američke financijske softverske kompanije Intuit. Sastoji se od jednostavnog i razumljivog sučelja na kojem se nalaze kategorizirane i vremenski dodijeljene financijske značajke. Glavne kategorije Mint-a su: pregled troškova, transakcije, računi, proračuni, financijski ciljevi, trendovi, ulaganje i štednja. Aplikacija automatski ažurira svaku od kategorija koristeći podatke s bankovnih računa i kreditnih kartica što i jest njezina najveća prednost [4]. Usluga integracije s bankarskim institucijama ograničena je na područje SAD-a i Kanade. Također, omogućuje korisniku personalizaciju kategorija, šalje obavijesti za plaćanje računa te predlaže ograničenja za pojedine kategorije usklađene s primanjima [5]. Aplikacija je besplatna i dostupna za iPhone, iPad i Android uređaje.

2.1.2. You Need a Budget (YNAB)

YNAB je aplikacija koja u fokus stavlja budžetiranje i kontrolu troškova. Filozofija na kojoj se zasniva je "svrha za svaki zarađeni dolar" [4]. Na taj način potiče korisnike da promišljaju unaprijed i odrede kategorije u kojima će se zarađeni novac alocirati: štednja, troškovi ili ulaganja. Uz navedene opcije korisnik može sam dodati detaljniju kategorizaciju i postaviti različite

financijske ciljeve. Nadalje, određeni budžeti mogu biti podijeljeni između više korisnika. U prvim verzijama aplikacije bilo je nužno ručno unošenje prihoda i rashoda, ali novije verzije sinkroniziraju se s bankovnim računima. Uporaba aplikacije se plaća na mjesečnoj ili godišnjoj razini, a od listopada 2019. godine cijena joj je 7$ mjesečno. YNAB je primarno web aplikacija, ali nudi i podršku za mobilne uređaje u Pro verziji (Android i iPhone) [5].

2.1.3. Wally

Wally je jednostavna aplikacija za praćenje troškova koja pruža uvid u potrošačke navike korisnika. Kategorija rashoda i njihov grafički prikaz predstavljaju osnovnu značajku aplikacije. Tome su dodane obavijesti o prekomjernom ili neplaniranom trošku, kao i o podijeljenim troškovima između više korisnika. Također, omogućuje spremanje fotografija računa i važnih financijskih dokumenata kako bi se u potpunosti izbjegla uporaba papira u praćenju osobnih financija [4]. Ne nudi mogućnost povezivanja s bankovnim računom što s aspekta sigurnosti mnogi korisnici smatraju poželjnim [6]. Aplikacija podržava unošenje podataka u gotovo svim valutama. Besplatna je i dostupna za Android i iPhone mobilne platforme.

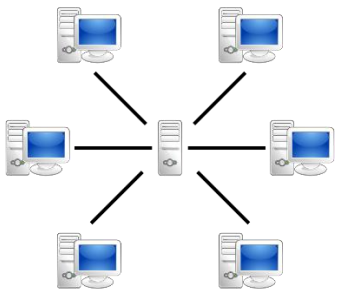
2.1.4. Toshl Finance

Toshl Finance je slovenska aplikacija nastala 2010. godine. Nudi jednostavno i zabavno korisničko sučelje uz prikaz animiranih likova koji korisnika vode kroz korištenje aplikacije. Kritičari je opisuju kao “sveobuhvatnu, praktičnu aplikaciju koja unosi lakoću u inače dosadne zadatke” [7]. Omogućuje analizu prošlih troškova po vremenskim intervalima, a istovremeno osigurava mogućnost kreiranja budućih budžeta. Osim iznosa, kategorije i datuma trošak je moguće unijeti i u raznim valutama. Aplikacija podržava 161 valutu, a ažuriranje tečaja je svakodnevno. Dolazi u osnovnoj verziji koja je besplatna, ali i u Toshl Pro obliku čija se uporaba naplaćuje. Aplikacija je dostupna kao web aplikacija te za sve mobilne platforme.

3. Arhitektura sustava

Raspodijeljeni sustav (engl. *distributed system*) definira se kao sustav čiji se hardverski ili softverski dijelovi nalaze na različitim čvorovima (pod čvorom se najčešće podrazumijeva računalo – poslužitelj) u mreži, a međusobno razmjenjuju poruke kako bi obavili zadani posao [8]. Može se reći da vrijedi tvrdnja da bi neki sustav bio nazvan raspodijeljenim, on mora svojim korisnicima izgledati jedinstven. Naime, raspodijeljenost je značajka samo na razini sustava, dok ona ne smije biti vidljiva na prezentacijskoj razini. Budući da se raspodijeljenost koristi dulje vremensko razdoblje, razvijeno je nekoliko arhitektura od kojih je svaka pogodna za određenu namjenu.

Arhitektura sustava u kojem je jedan od sudionika poslužitelj, dok drugi predstavlja njegovog klijenta jedan je od prvih načina korištenih u svijetu raspodijeljenih sustava. Osnovna klijent-poslužitelj arhitektura prikazana je na slici (Slika 3.1.). Klijenti u ovakvom sustavu raspolažu aplikacijom koja se spaja na poslužitelja i koja je u mogućnosti interpretirati podatke koje dobiva od poslužitelja. S druge strane, na poslužitelju se odvija proces koji stalno osluškuje i prihvaća zahtjeve klijenata, obrađuje ih i vraća im rezultate (Ljubi, 2013; Mihaljević, 2013). Poslužitelj također može dohvaćati podatke i nekog drugog poslužitelja, postajući tako njihov klijent.



Slika 3.1. Shema arhitekture klijent-poslužitelj[1]

[1] https://bs.wikipedia.org/wiki/Server_(ra%C4%8Dunarstvo), prosinac 2019.

Klijent-poslužitelj arhitektura čini osnovu razvijenog cjelovitog raspodijeljenog sustava za kontrolu i vođenje osobnih financija. Sastoji se, dakle, od poslužitelja koji predstavlja jedinstveni izvor podataka te web i mobilne aplikacije kao klijenata. Klijenti se povezuju s navedenim izvorom i dohvaćaju potrebne podatke.

U ovakvoj arhitekturi klijent uglavnom nije niti svjestan koliko se zapravo poslužitelja krije u pozadini koji osiguravaju ispravan i nesmetan prikaz informacija koje je zatražio. Postoji više tipova klijenata, a u ovom razvijenom sustavu klijenti su debeli (engl. *fat client*). Prema definiciji oni su samostalni aplikacijski klijenti koji se izvode na klijentskim računalima veće snage, a omogućuju funkcionalnosti bogatog grafičkog korisničkog sučelja. Sadrže i napredne mogućnosti kao što su povuci-i-spusti (engl. *drag-and-drop*), kontekstno osjetljivi dijelovi prikaza, iscrtavanja vektorske 2D i 3D grafike i slično. Interoperabilnost informacijskog sustava izvedena je na razini klijentskog sloja prema poslovnoj logici (Ljubi, 2013; Mihaljević, 2013).

Poslužitelj je implementiran uporabom Web API REST arhitekturalnog modela u ASP.NET Core 3.0 (u nastavku .NET Core 3) tehnologiji. U potpoglavlju 3.2.4. detaljnije su objašnjeni principi rada REST arhitekturalnog modela. U trenutku razvoja sustava 3.0 verzija je bila najnovija dostupna verzija .NET Core programskog okvira. To razvijeni sustav čini aktualnim u smislu praćenja tehnoloških trendova. Poslužitelj služi kao aplikacijsko programsko sučelje za prikaz podataka iz baze podataka klijentskim aplikacijama. U tako postavljenoj arhitekturi klijenti nemaju direktan pristup bazi podataka, nego sve tražene resurse dohvaćaju kroz API na koji se povezuju. Također, korisnicima su dostupni jednaki podaci neovisno o tipu klijenta kojim se spajaju na poslužitelja.

Web klijent izrađen je u Vue.js programskom okviru u *Single Page Application* (skraćeno SPA) tehnologiji. Mobilni klijent izveden je u Nativescript-Vue programskom okviru koji je dizajniran za rad na svim mobilnim platformama. To sustav čini dostupnim velikom broju korisnika tj. svima koji imaju Internet preglednik i računalo ili pametni telefon.

U izradi sustava korištena je inkrementalna metodologija razvoja softvera. Navedena metodologija dio je životnog ciklusa razvoja softvera (engl. *Software development life cycle*, skraćeno SDLC), a označava razvoj sustava u nizu iteracija, a ne odjednom. Obično se implementira mali podskup traženih funkcionalnosti koje se mogu prikazati korisniku. Sustav se razvija konstantno kroz nadogradnju i daljnje dodavanje funkcionalnosti dok ne postane u potpunosti spreman za

implementaciju. Takav razvoj softvera obično se koristi u situacijama kada su glavni zahtjevi korisnika u potpunosti jasni i definirani, ali određene funkcionalnosti ili nadogradnje se mogu naknadno definirati.

3.1. Baza podataka

Baza podataka je skup tablica koje sadrže neke podatke. Tablice s podacima su pohranjene na vanjskoj memoriji, koji su istovremeno dostupne korisnicima i programima [9]. Sastoji se od skupa međusobno povezanih podataka, pohranjenih zajedno, bez štetne ili nepotrebne zalihosti. Podaci su u bazi pohranjeni u obliku neovisnom o aplikacijama koje ih koriste [10]. Danas je teško zamisliti aplikaciju ili poslovni sustav bez baze podataka. U stvarnom poslovnom svijetu ona predstavlja osnovni element svakog korisničkog informacijskog sustava. Primarna svrha joj je pohranjivanje, skladištenje i upravljanje podacima vitalnim za rad informacijskog sustava.

Podaci u bazi logički su organizirani u skladu s nekim modelom podataka. Model podataka je skup pravila koja određuju kako može izgledati logička struktura baze. Pri izradi sustava upotrijebljena je **Microsoftova** baza podataka koja primjenjuje relacijski model, zasnovan na matematičkom pojmu relacije. Podaci i veze među podacima prikazuju se tablicama. Tablice su međusobno povezane relacijama čija je uloga izbjeći redundanciju (ponavljanje), očuvati integritet podataka, te povećati brzinu pretraživanja [11].

3.1.1. Entity Framework Core

Entity Framework Core (skraćeno EF Core) je posljednja verzija popularne Microsoft-ove Entity Framework tehnologije za pristup podacima iz baze podataka. Prednosti njegovog dizajna su što je otvorenog kôda (engl. *open source*), lakoća, mogućnost proširivanja i podrška za više razvojnih platformi, kao i .NET Core programski okvir. Jednostavan je za uporabu i nudi performansna poboljšanja u odnosu na prijašnje verzije Entity Framework-a. Poglavito služi kao *object-relational-mapper* (skraćeno ORM). ORM je tehnika koja omogućuje rad s podacima na objektno-orijentirani način, izvršavajući mapiranje objekata iz programskog kôda u objekte spremljene u

bazi podataka i obratno [12]. Nadalje, eliminira potrebu pisanja dodatnog kôda za pristup bazi podataka te tu logiku odrađuje samostalno. Iako je Microsoft-ov proizvod, ne podržava pristup samo njihovim, nego i ostalim popularnim bazama podataka [13].

Dodatne prednosti rada s EF Core-om su sljedeće:

- kreira bazu podataka i održava njezinu shemu u skladu s promjenama na modelima po kojima je napravljena
- generira *Structured Query Language* (skraćeno SQL) i izvršava ga nad bazom
- upravlja transakcijama
- upravlja objektima koji su već dohvaćeni iz baze [12].

U EF Core-u pristup podacima iz baze izvršava se preko modela. Model se sastoji od entitetskih klasa, konfiguracije Entity Framework-a i objekta konteksta (engl. *context*). Objekt konteksta predstavlja sesiju s bazom podataka i omogućuje izvedbu operacije nad podacima (spremanje, brisanje i uređivanje) [13]. Model se može generirati iz postojeće baze podataka i takav pristup naziva se *database-first-approach*. Popularan je u informacijskim sustavima gdje već postoji baza podataka ili ju je kreirao administrator baza podataka i ima posebna pravila i ograničenja. Postoji i *code-first-approach* u kojem programer razvija model kôdom kreirajući klase koje predstavljaju entitete i putem EF migracija kreira bazu podataka. Migracije predstavljaju način kojim se shema baze podataka usklađuje s modelima koje programer koristi. Kada se entitetske klase u modelu promijene na bilo koji način, potrebno je pokrenuti migraciju kako bi EF Core API ponovno kreirao model i prikupio napravljene promjene. Primijenivši novonastalu migraciju s promjenama nad bazom, ona dolazi u stanje jednakosti s modelom u kôdu. Svaka promjena baze odrađuje se migracijama u kôdu čime je izbjegnuto ručno ili neautorizirano mijenjanje baze. U takvom pristupu baza evoluira zajedno s modelom i promjenama kojim programer mijenja model kroz vrijeme. Time se zadržava potpuna kontrola nad bazom.

U ovom radu korišten je Entity Framework Core verzije 3.0.0-preview6.19307.2 te *code-first-approach*. U početku nije postojala baza jer se činilo da će takav pristup točno odgovarati potrebama razvijenog sustava.

Konfiguracija je jedna od sastavnica modela Entity Framework-a. Definira se u klasi *Startup.cs* Web API projekta, a prikazana je u kôdu (Kôd 3.1.):

```
services.AddDbContext<ExpensioContext>(config =>
      {
      config.UseSqlServer(Configuration.GetConnectionString("ExpensioDB"),
      options=> {
      options.MigrationsAssembly("Expensio");
      });
});
```

Kôd 3.1. Konfiguracija Entity Framework-a

Konfiguriran je naziv konteksta za bazu podataka – *ExpensioContext* i uporaba SQL Servera kao relacijske baze podataka. Navedenom je podešen i konekcijski string (engl. *connection string*) za povezivanje na bazu. Opcija na *options* objektu „MigrationsAssembly“ služi za definiciju modula u kojem se nalaze migracije za određeni kontekst [14].

Budući da je projekt podijeljen u više različitih modula, a *ExpensioContext* se nalazi u modulu *Expensio.Infrastructure*, bilo je potrebno definirati *Expensio* modul kao onaj modul koji sadrži migracije.

3.1.2. Inicijalno postavljanje baze

Baza podataka razvijenog sustava sastoji se od ukupno šesnaest tablica. Devet tablica je auto-generirano od strane Microsoft Identity sustava koji će biti objašnjen u Poglavlju 4.1. Ostalih sedam tablica modelirane su direktno kôdom i predstavljaju entitete vitalne za funkcioniranje sustava.

Nakon kreiranja projekta napravljena je inicijalna migracija. Pomoću migracije, Microsoft Identity kreira svoje tablice u bazi u koje se spremaju podaci potrebni za autentikaciju korisnika. U tablicama su spremljeni podaci o korisnicima, uloge koje se koriste u sustavu te ostali podaci (npr.

način na koji se korisnik prijavljuje u sustav). Uporabom Microsoft Identity-a nije nužno korištenje svih tablica koje su kreirane iako one svejedno postoje. Jedna od njihovih svrha može biti naknadno korištenje pri nadogradnji sustava.

Migracije se dodaju putem naredbe „*Add-Migration*“ u *Package Manager Console*-i (skraćeno PMC). PMC je *Powershell* konzola dostupna unutar razvojnog alata Visual Studio i služi za instalaciju NuGet paketa. Za dodavanje migracije potrebno je instalirati NuGet biblioteku *Microsoft.EntityFrameworkCore.Tools* unutar PMC-a. Pomoću tog alata kreiraju se migracije, primjenjuju se iste u bazi ili se generira kôd za modele temeljen na postojećoj bazi (*database-first-approach*). Naredbe se izvršavaju unutar Visual Studio-a i njegovog PMC-a. Za dodavanje inicijalne migracije upotrijebljena je naredba „*Add-Migration CreateIdentitySchema*“. Automatski je kreiran direktorij „*Migrations*“ unutar projekta i u njega je spremljena inicijalna migracija. Direktorij će u nastavku služiti kao centralno mjesto na kojemu se pohranjuju sve migracije vezane za projekt. U tom trenutku migracija još nije primijenjena na bazi, ali je dostupna programeru u slučaju potencijalnih preinaka. Nakon pregleda migracije izvršena je naredba „*Update-Database*“ u PMC koja označava primjenu migracije. U slučaju inicijalne migracije to podrazumijeva kreiranje sedam tablica potrebnih Microsoft Identity-u za funkcioniranje.

Nakon definiranja potrebnih entitetskih modela i dodavanja ostalih migracija, pokrenuta je naredba „*Update-Database*“ i dobivena je potpuna baza podataka. Ovakav pristup u razvoju sustava omogućuje programeru brzu kreaciju baze, tablica i ostalih potrebnih objekata. Međutim, prilikom puštanja aplikacije u produkciju primjenjuju se drugačije metode, a jedna od poznatijih je generiranje SQL skripte iz postojećih migracija, dobivena pomoću naredbe „*Script-Migration*“. Takva skripta podrazumijeva da je baza podataka u produkciji kreirana te u njoj stvara tablice i sve ostale objekte koji se nalaze u migracijama. Skriptu je moguće doraditi tako da sama stvori bazu, korisnike (engl. *user*) i ostale potrebne objekte koji su korišteni za vrijeme razvoja sustava. Kao takva predstavljala bi potpuno samostalnu cjelinu i mogla bi kreirati sve što je potrebno za rad sustava [14].

Radi pravilnog funkcioniranja sustava i jednostavnijeg testiranja dodane su kategorije troškova, vrste prihoda, određene korisničke uloge i jedan testni korisnik s administratorskim pravima. Budući da se radi o *code-first-approach*-u kod kreiranja baze podataka, sve to učinjeno je u kôdu.

EF Core sadrži metodu *OnModelCreating*() u *ExpensioContext*-u koja se koristi za postavljanje inicijalnih podataka.

```
protected override void OnModelCreating(ModelBuilder builder)
{
	base.OnModelCreating(builder);
	builder.Entity<IncomeType>().HasData(
		new IncomeType() { IncomeTypeId = 1, Name = "Salary" },
		new IncomeType() { IncomeTypeId = 2, Name = "Awards" },
		new IncomeType() { IncomeTypeId = 3, Name = "Rent" }
	);
	builder.Entity<Category>().HasData(
		new Category() { CategoryId = 1, CategoryName = "Rent" },
		new Category() { CategoryId = 2, CategoryName = "Gas" },
		new Category() { CategoryId = 3, CategoryName = "Electricity" },
		new Category() { CategoryId = 4, CategoryName = "Internet" },
		new Category() { CategoryId = 5, CategoryName = "Water" },
		new Category() { CategoryId = 6, CategoryName = "Garbage" },
		new Category() { CategoryId = 7, CategoryName = "Food" },
		new Category() { CategoryId = 8, CategoryName = "Allowance" },
		new Category() { CategoryId = 9, CategoryName = "Fun" },
		new Category() { CategoryId = 10, CategoryName = "Appliances" }
	);
	builder.Entity<AccountType>().HasData(
		new AccountType() { AccountTypeId = 1, Type = "Single" },
		new AccountType() { AccountTypeId = 2, Type = "Shared" }
	);
	builder.Entity<Account>().HasData(
		new Account() { AccountId = 1, AccountTypeId = 1 }
	);
}
```

Kôd 3.2. Izgled metode *OnModelCreating()* u *ExpensioContext*-u

Kao što je vidljivo u kôdu (Kôd 3.2.), inicijalni podaci o korisniku nisu postavljeni u ovoj metodi. Microsoft Identity u kombinaciji s ASP.NET Core-om zahtijeva drugačiji pristup. U metodi *Configure()* u *Startup.cs* klasi dodaju se dodatna dva parametra tipa *UserManager* i *RoleManager* te se poziva metoda *SeedData()* na klasi *MyIdentityDataInitializer*:

```
public void Configure (…UserManager<ApplicationUser> userManager,
RoleManager<ApplicationRole> roleManager)
{
      …
      MyIdentityDataInitializer.SeedData(userManager, roleManager);
}
```

UserManager i *RoleManager* predstavljaju API Microsoft Identity-a za upravljanje korisnicima i ulogama u sustavu. Njihovim prosljeđivanjem u metodu *SeedData()* osigurana je uporaba upravo tih objekata tijekom stvaranja korisnika i uloga (engl. *role*). Umjesto konfiguracije u metodi *Configure()*, konfiguracija je izolirana u posebnu klasu i pozvana je od tamo [15].

U kôdu (Kôd 3.3.) je prikazana konkretna implementacije metode *SeedRoles()*.

```
public static void SeedRoles(RoleManager<ApplicationRole> roleManager)
{
      if (!roleManager.RoleExistsAsync("User").Result)
      {
            ApplicationRole role = new ApplicationRole();
            role.Name = "User";
            role.Description = "Perform normal user operations.";
            _ = roleManager.CreateAsync(role).Result;
      }
      if (!roleManager.RoleExistsAsync("Administrator").Result)
      {
            ApplicationRole role = new ApplicationRole();
            role.Name = "Administrator";
            role.Description = "Perform all the operations.";
            _ = roleManager.CreateAsync(role).Result;
      }
}
```

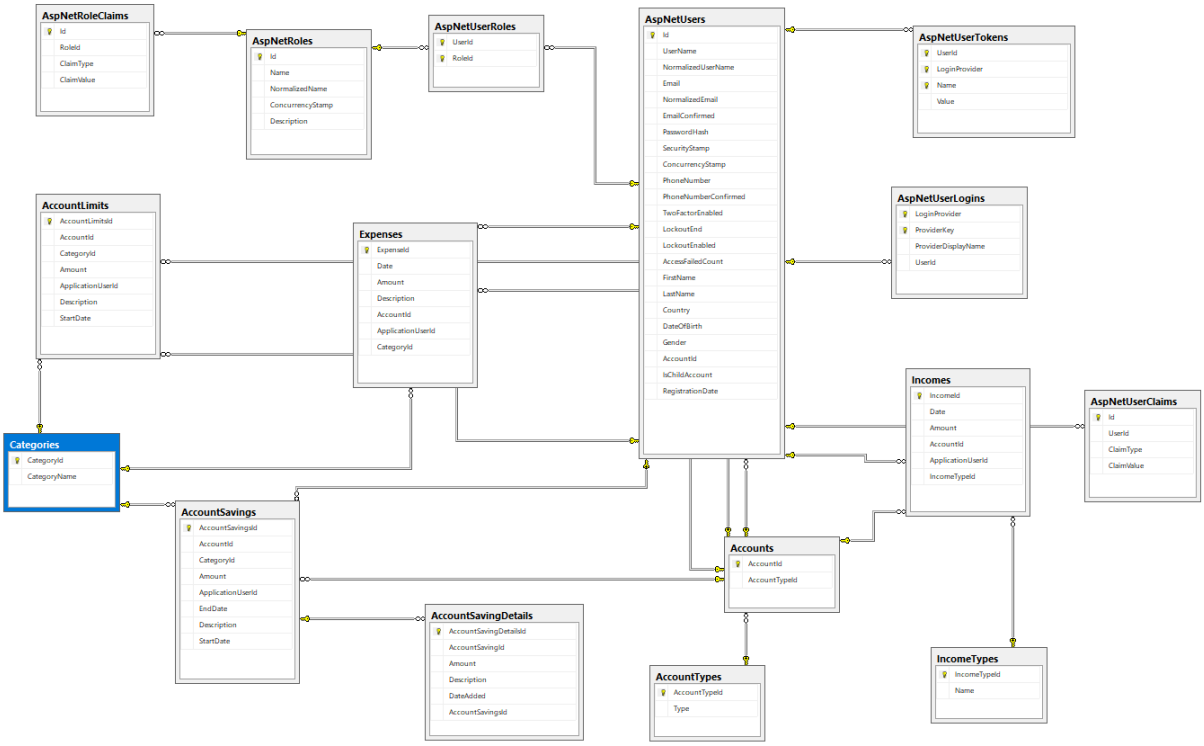
Kôd 3.3. Implementacija metode *SeedRoles()*

Metoda *SeedData()* poziva dvije statičke metode – *SeedRoles()* i *SeedUsers()*. *SeedRoles()* kreira željene uloge u sustavu pozivom metode *CreateAsync()* nad proslijeđenim objektom *roleManager*.

SeedUsers() kreira inicijalnog korisnika, također, pozivom metode *CreateAsync()* nad proslijeđenim objektom *userManager*. Metodu *SeedRoles()* potrebno je pozvati prvu kako bi se osiguralo postojanje uloge u trenutku dodjeljivanja kreiranom korisniku.

3.1.3. Tablice i entiteti

Slika (Slika 3.2.) predočava ER dijagram potpuno kreirane baze podataka sa svim tablicama.



Slika 3.2. ER dijagram baze podataka

U nastavku (Tablica 3.1.) se nalaze objašnjenja tablica generiranih iz entiteta te korištenih tablica generiranih od Microsoft Identity-a.

Tablica 3.1. Tablice iz baze podataka

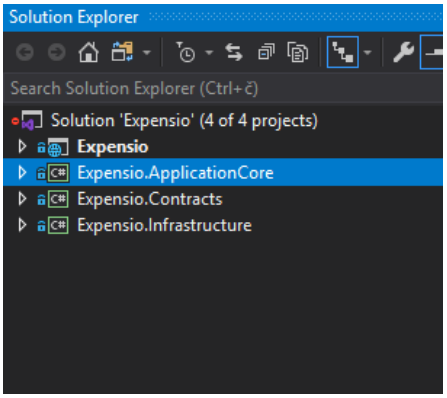
TABLICA	OPIS

AccountLimits	Ograničenja koja korisnik postavlja na račun. Primarni ključ je *AccountLimitsId,* a uključuje kolone poput iznosa limita, opisa i datuma početka.
Accounts	Svaki korisnik pri kreiranju dobije svoj račun. Ova tablica prikazuje popis računa koji postoje u sustavu i njihov tip. Postoje samo dvije kolone, *AccountId* i *AccountTypeId*.
AccountSavings	Štednja po korisničkom računu. Svaki korisnik može imati više štednji ili omotnica. Primarni ključ je *AccountSavingsId,* a sadrži podatke o datumu početka i kraja štednje, iznosu, opisu itd.
AccountSavingDetails	Mjesto pohrane detalja o uplati u određenu štednju. Odnos je 1:N s tablicom *AccountSavings* gdje jedna štednja može imati više „uplata“. Postoje kolone poput datuma dodavanja i iznosa kako bi se mogao pratiti ukupno uplaćeni iznos u štednju u odnosu na postavljeni cilj.
AccountTypes	Šifrarnik koji predstavlja tipove računa koji postoje. Trenutno su dodane dvije opcije, *single* i *shared*.
Expenses	Glavna tablica svih korisničkih troškova zabilježenih u sustav. Primarni ključ joj je *ExpenseId,* a troškove korisnika razlikuje po poljima *ApplicationUserId* i *AccountId*. Sadrži

	podatke poput iznosa troška, kategorije troška, opisa i datuma kreiranja.
Categories	Šifrarnik mogućih kategorija troškova. Sadrži samo dvije kolone, *CategoryId* koja je ujedno i primarni ključ i *CategoryName*.
Incomes	Glavna tablica svih korisničkih prihoda u sustavu. Primarni ključ je *IncomeId,* a postoje još i podaci o iznosu, datumu i tipu prihoda te korisnički podaci poput *ApplicationUserId* i *AccountId* po kojima se razlikuju.
IncomeTypes	Šifrarnik mogućih tipova prihoda. Sadrži samo dvije kolone, *IncomeTypeId* koja je ujedno i primarni ključ i *Name*.
AspNet*Users*	Glavna tablica Microsoft Identity-a za pohranu podataka o korisniku. Sadrži osobne podatke poput imena, prezime, adrese i e-mail-a, kao i korisničko ime, lozinku, te broj računa (engl. *account*).
AspNet*Roles*	Šifrarnik mogućih uloga (engl. *roles*) u sustavu. Najvažnije kolone su ime uloge i opis.
AspNetUser*Roles*	Vezna tablica između *AspNetUsers* i *AspNetRoles*; sadrži dodijeljene uloge korisnika.

3.2. Poslužitelj

Poslužitelj je kao projekt postavljen u razvojnom alatu Visual Studio, strukturno podijeljen u slojeve što se vidi na slici (Slika 3.3). Glavni definirani sloj, naziva *Expensio*, predstavlja Web API. Nadalje, postoje slojevi *Expensio.ApplicationCore* koji predstavlja servisni sloj, *Expensio.Infrastructure*; sloj pristupa podacima u kojem se nalazi konekcija s bazom podataka te *Expensio.Contracts* gdje su definirani korišteni entitetski modeli.

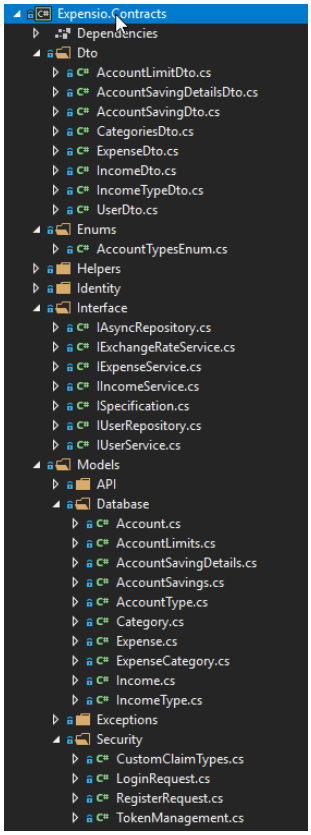


Slika 3.3. Strukturni slojevi poslužitelja

3.2.1. Sloj jezgre

Sloj jezgre (engl. *core layer*) čini *Expensio.Contracts* sloj. U njemu se nalaze svi modeli koji su korišteni u sustavu. Sadrži kreirane entitetske klase koje čine model za Entity Framework Core, a koje su kreirane koristeći *code-first-approach*. Sadrži definicije sučelja koja će biti implementirana u *Expensio.ApplicationCore* sloju. Tu se nalaze i klase *ApplicationRole* i *ApplicationUser* koje su prilagođene klase Microsoft Identity-a. O njima će više govora biti u poglavlju 4. Sadrži i sve objekte za prijenos podataka (engl. *Data Transfer Object*, DTO). Svrha tih objekata je slanje podataka iz baze podataka na prezentacijski sloj bez izlaganja same entitetske klase koje predstavljaju tablice. Svaki objekt koji predstavlja podatak iz baze podataka je potrebno mapirati na odgovarajući DTO.

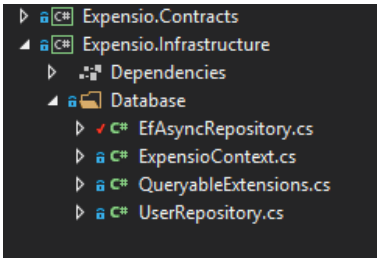
Struktura sloja jezgre prikazana je na slici (Slika 3.4.).



Slika 3.4. Struktura jezgre

3.2.2. Infrastrukturni sloj

Infrastrukturni sloj predstavlja sloj poslužitelja koji direktno komunicira s bazom podataka koristeći Entity Framework Core, a njegova struktura vidljiva je na slici (Slika 3.5.). U troslojnim aplikacijskim arhitekturama ovo se tipično naziva sloj pristupa podacima (engl. *Data Access Layer*, skraćeno DAL). DAL predstavlja apstrakciju nad bazom podataka kako bi se teoretski mogao promijeniti izvor podataka na npr. neku drugu bazu, uz neometani rad aplikacije. Također, predstavlja apstrakciju nad logičkim modelom podataka u smislu slabog povezivanja poslovnog sloja aplikacije s DAL slojem i uporabom sučelja za komunikaciju, a ne konkretne implementacije klase.



Slika 3.5. Struktura infrastrukturnog sloja

Glavna klasa koju sadrži je *ExpensioContext*, predočena je sljedećim kôdom (Kôd 3.4.).

```
public class ExpensioContext :
IdentityDbContext<ApplicationUser,ApplicationRole, int>
{
      public DbSet<Income> Incomes { get; set; }
      public DbSet<IncomeType> IncomeTypes { get; set; }
      public DbSet<Expense> Expenses { get; set; }
      public DbSet<Category> Categories { get; set; }
      public DbSet<Account> Accounts { get; set; }
      public DbSet<AccountSavings> AccountSavings { get; set; }
      public DbSet<AccountLimits> AccountLimits { get; set; }
      public DbSet<AccountType> AccountTypes { get; set; }
      public DbSet<AccountSavingDetails> AccountSavingDetails { get; set; }
            …
}
```

Kôd 3.4. Klasa *ExpensioContext*

U klasi se nalaze svi DbSet-ovi koji predstavljaju kolekciju za određenu entitetsku klasu unutar EF Core modela. Služe kao poveznica za izvršavanje operacija u bazi podataka nad tim entitetom. *DbSet<T>* klase su dodane kao svojstva *ExpensioContext*-u. Mapiraju se na tablice u bazi podataka po imenu, gdje ime svojstva ujedno predstavlja i ime tablice [16].

U ovom sloju također se nalazi generička implementacija *EfAsyncRepository* prikazana kôdom (Kôd 3.5.) koja implementira generičko sučelje *IAsyncRepository* koje se nalazi u sloju jezgre, a prikazano je kôdom (Kôd 3.6.).

```
public class EfAsyncRepository<T> : IAsyncRepository<T> where T : class
{
	protected ExpensioContext DbContext;
	public EfAsyncRepository(ExpensioContext context)
	{
		DbContext = context;
	}
		…
}
```

Kôd 3.5. Generička implementacija *EfAsyncRepository* klase

```
public interface IAsyncRepository<T> where T : class
{
	Task<T> GetById(int id);
	Task<IEnumerable<T>> GetAll(TrackingOption tracking =
	TrackingOption.WithTracking);
	Task<IEnumerable<T>> GetByCondition(Expression<Func<T, bool>> expression);
	Task<int> Add(T entity);
	Task AddRange(IEnumerable<T> entities);
	Task<int> Update(T entity);
	Task<int> Delete(T entity);
	Task DeleteRange(IEnumerable<T> entities);
}
```

Kôd 3.6. Generičko sučelje *IAsyncRepository*

U kôdu (Kôd 3.5.) vidljiva je uporaba dizajna principa ubrizgavanja ovisnosti (engl. *dependency injection*), s ciljem ubrizgavanja *ExpensioContext* implementacije EF Core kontekstne klase. Ubrizgavanje ovisnosti je princip koji predstavlja dostavljanje zavisne klase kroz svojstvo, konstruktor ili metodu [17].

Arhitekturalno repozitorij nije implementiran u obrascu repozitorija (engl. *Repository pattern*) iako se zbog generičke implementacije *EfAsyncRepository.cs* možda čini tako. Razlog tome je u sljedećem objašnjenju.

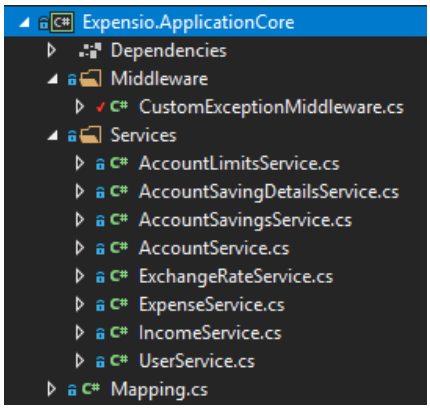
Povijesno najčešća svrha korištenja obrasca repozitorija je bila pružiti dodatan sloj apstrakcije nad Entity Framework-om. To rješenje nije bilo dobro za svaku strategiju pristupa podacima iz baze podataka, iako je u određenim kontekstima ipak bilo najbolje rješenje. Prije nekoliko godina, izlazak prve verzije Entity Framework-a, koja nije bila pogodna za pisanje testova dijelova (engl. *unit tests*), izazivao je frustraciju korisnika/programera. Tada je započelo apstrahiranje Entity Framework-a iza obrazaca repozitorija i jedinica-rada (engl. *Unit-Of-Work pattern*), kako bi se, među ostalim, olakšalo testiranje. Razvojem Entity Framework-a, posebno EF Core-a, njegova uporaba postala je puno jednostavnija, a potreba za apstrahiranjem radi pisanja testova dijelova izostala. Potrebno je istaknuti da je *DbContext* unutar Entity Framework-a primjer obrasca jedinica-rada, a *DbSet<T>* već je obrazac repozitorija koji pruža apstrakciju nad slojem za pristup podataka.

Još jedna ideja apstrahiranja Entity Framework-a iza obrasca repozitorija i jedinice-rada bila je ubrizgavanje repozitorija/servisa kako se *DbContext* ne bi morao direktno ubrizgavati u upravitelje (engl. *controller*). Tako je omogućena izrada slabo povezanih aplikacija (engl. *loosely coupled*). Istovremeno se postavilo pitanje što će se dogoditi u budućnosti u slučaju moguće zamjene ORM-a s kojim se spaja na bazu. Manjkavost takvog pristupa bila je u tome što se uglavnom slučaj mijenjanja ORM-a rijetko događao, a čak i ako je postojale su druge metode i bolji obrasci za otklanjanje tog potencijalnog budućeg rizika.

Bez obzira na to što EF Core čini obrasce repozitorija i jedinice rada nepotrebnim, nije najbolje rješenje ubrizgavati *DbContext* EF Core-a direktno u upravitelje. Razlog tome nije u otežavanju potencijalne promjene ORM-a u budućnosti, nego u zadržavanju upravitelja „čistima“, malima i fokusiranima na vraćanje podataka prezentacijskom sloju. Tipični primjer toga je transformacija entiteta iz baze podataka u objekte za prijenos podataka koja je čest zahtjev u poslovnim okruženjima. Te operacije su prikladnije za obrazac servisnog sloja (engl. *Service Layer Pattern*) koji je u konačnici i korišten u razvijenom sustavu [18].

3.2.3. Servisni sloj

Prema definiciji, servisni sloj definira granice unutar aplikacije i postavlja set dozvoljenih operacija iz perspektive klijentskog sloja. Enkapsulira aplikacijsku poslovnu logiku, upravljanje transakcijama i koordinaciju odgovora u implementacijama svojih metoda [19]. Drugim riječima, servisni sloj služi kao posrednik između API i infrastrukturnog sloja, a glavna uloga mu je dohvaćanje podataka iz infrastrukturnog sloja, mapiranje entitetske klase u odgovarajuće DTO modele i vraćanje istih pozivatelju iz API sloja. Struktura servisnog sloja poslužitelja prikazana je na slici (Slika 3.6.).



Slika 3.6. Struktura servisnog sloja

U srži ovog sloja nalaze se specifične implementacije servisnih sučelja definiranih u sloju jezgre. Svaki servis ima svoje odgovarajuće servisno sučelje koje definira metode, a koje servis mora implementirati. Također, svaki servis nasljeđuje klasu *EfAsyncRepository.cs* (Kôd 3.5.) koja je implementacija generičkog asinkronog repozitorija i njoj predaje svoju entitetsku klasu kao generički tip *T*. Servisi imaju i *ExpensioContext* implementaciju direktno ubrizganu kao ovisnost te se na taj način ostvarila uska povezanost (engl. *tightly coupled*) sa slojem pristupa podacima. Time je svaki servis dobio mogućnost pozivanja osnovnih metoda koje taj generički repozitorij implementira nad predanim entitetom. Omogućeno je dohvaćanje podataka iz baze podataka ili direktni pristup bazi podataka kroz objekt *context* ukoliko postoji potreba za dodatnim operacijama koje nisu definirane u generičkom repozitoriju. Takav način arhitekture odabran je iz razloga objašnjenih u prethodnom potpoglavlju. Jedina iznimka u odnosu na ovaj pristup je *UserService.cs*

koji upravlja operacijama nad korisnicima te isključivo koristi *UserRepository.cs* ubrizgan kao ovisnost. On opet koristi API Microsoft Identity-a za pohranu podataka o korisnicima te mu nije potrebna generička implementacija repozitorija.

Kôd (Kôd 3.7.) prikazuje servis *IncomeService.cs* i njegovu ovisnost o *ExpensioContext*-u te nasljeđivanje klase *EfAsyncRepository.cs*. Može se uočiti i vraćanje DTO modela iz metode *GetAllUserIncomes()* što odgovara ulozi servisnog sloja kao transformatora podataka.

```
public class IncomeService : EfAsyncRepository<Income>, IIncomeService
{
	private ExpensioContext context;
	public IncomeService(ExpensioContext context) : base(context)
	{
		this.context = context;
	}

	public Task<List<IncomeDto>> GetAllUserIncomes(int userId)
	{
	var incomes = context.Incomes.Where(x => x.ApplicationUser.Id ==
	userId).Include(x => x.IncomeType).Include(x => x.ApplicationUser);
	var incomesDto = incomes.AsEnumerable().Select(y =>
	Mapping.Mapper.Map<IncomeDto>(y));
	return Task.FromResult(incomesDto.ToList());
	}

	public async Task<bool> SaveNewUserIncome(IncomeDto income)
	{
	var modelForDB = Mapping.Mapper.Map<Income>(income);
	var result = await this.Add(modelForDB);
	return result == 1 ? true : false;
	}
}
```

Kôd 3.7. Servis *IncomeService.cs*

Servisni sloj sadrži i *CustomExceptionMiddleware.cs*. Kôd (Kôd 3.8.) predočuje klasu koja se bavi dohvaćanjem i obrađivanjem iznimke (engl. *Exception*) u kôdu na globalnoj tj. aplikacijskoj razini. *Middleware* u ASP.NET Core-u predstavlja dio softvera koji obrađuje zahtjeve i odgovore unutar aplikacijskog tijeka izvršavanja (engl. *pipeline*). Time je omogućeno brisanje svih *try-catch* blokova iz kôda što je rezultiralo čišćim i jasnijim kôdom [20].

```
public class CustomExceptionMiddleware
{
	private readonly RequestDelegate _next;
	public CustomExceptionMiddleware(RequestDelegate next)
	{
		_next = next;
	}

	public async Task InvokeAsync(HttpContext httpContext)
	{
		try
		{
		   await _next(httpContext);
		}
		catch (Exception ex)
		{
		   await HandleExceptionAsync(httpContext, ex);
		}
	}

	private Task HandleExceptionAsync(HttpContext context, Exception
	exception)
	{
		context.Response.ContentType = "application/json";
	context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;
		var error = new ErrorDetails()
		{
		   StatusCode = context.Response.StatusCode,
		   Message = exception.Message
		}.ToString();
```

```
            Console.WriteLine(error);
            return context.Response.WriteAsync(error);
        }
}
```

Kôd 3.8. Klasa za dohvaćanje i obrađivanje iznimke

U kôdu je (Kôd 3.8.) vidljivo da ukoliko se pojavi iznimke, ona se zapisuje u konzolu i nakon toga vraća pozivatelju kako bi na *frontend* strani mogla biti obrađena na određeni način. To u praksi nije slučaj i umjesto toga bi se vjerojatno našao poziv klase za zapisivanje (engl. *logging*) koja bi tu iznimku s dodatnim detaljima zapisala u datoteku ili poslala na udaljenog poslužitelja konfiguriranog isključivo za obrađivanje i spremanje iznimaka.

U servisnom sloju nalazi se i statična klasa *Mapping.cs* koja predstavlja konfiguraciju biblioteke *AutoMapper*. Upotrijebljena je za mapiranja entitetskih klasa vraćenih od strane EF Core-a u objekte za prijenos podataka (DTO). Uporabom *AutoMapper*-a izbjegnuto je ručno pisanje transformacija objekata koje se često pišu u entitetskim klasama.

3.2.4. API sloj

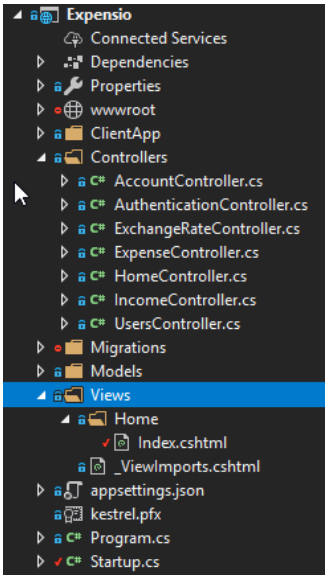
Sloj aplikacijskog programskog sučelja (skraćeno API) predstavlja sučelje za spajanje na poslovnu logiku sustava. Ovaj sloj implementiran je u REST arhitekturalnom modelu.

Representational State Transfer (skraćeno REST) je arhitekturalni model koji osigurava standard komunikacije između različitih računalnih sustava na mreži. U takvom arhitekturalnom modelu klijent i poslužitelj mogu biti implementirani u bilo kojoj tehnologiji bez da znaju jedno za drugo ili imaju ikakvih dodirnih točaka. Dok god znaju koji format poruka trebaju kako bi međusobna komunikacija bila uspješna, mogu biti razdvojeni. Korištenjem REST arhitekture više različitih klijenata se može spajati na istog poslužitelja, tražiti iste resurse, slati iste podatke i dobivati iste odgovore [21].

RESTful sustavi prate određene setove pravila kako bi se mogli nazvati *RESTful*. Okarakterizirani su kao sustavi koji ne čuvaju svoje prijašnje stanje (engl. *stateless*), a podijeljeni su na klijenta(e)

i poslužitelja. Obično komuniciraju putem HTTP(S) protokola kao najraširenijeg komunikacijskog protokola. Definirana su dva podatka koja klijent mora pružiti poslužitelju:

- Identifikator resursa za kojeg je zainteresiran. To se skraćeno naziva URL (engl. *Uniform Resource Locator*) ili *endpoint* na koji se klijent spaja.
- HTTP metoda koja označava operaciju za koju klijent želi da poslužitelj izvrši nad resursom. Najpoznatije metode su dohvati (engl. *Get*), spremi (engl. *Post*), ažuriraj (engl. *Put*) i obriši (engl. *Delete*) [22].



Slika 3.7. Sadržaj API sloja

Slika (Slika 3.7.) prikazuje sadržaj API sloja. Sadrži Web API upravitelje, migracije EF Core-a i modele za definiranje klasa za postavke aplikacije. Također, sadrži Vue.js klijentsku aplikaciju koja se nalazi u *ClientApp* direktoriju, a komunicirat će s Web API upraviteljima. Način njihove komunikacije bit će posebno objašnjen u Potpoglavlju 3.3.1. Također, sadrži i *Program.cs* klasu koja pokreće poslužitelja i *Startup.cs* klasu koja konfigurira servise korištene unutar aplikacije i njezin tijek izvršavanja (engl. *pipeline*).

```
[Authorize]
[ApiController]
[Route(„api/[controller]“)]
public class IncomeController : ControllerBase
{
      private readonly IIncomeService incomeService;

      public IncomeController(IincomeService incomeService)
      {
            this.incomeService = incomeService;
      }


      [HttpGet(„{userId:int:min(1)}/all“)]
      public async Task<ActionResult<List<IncomeDto>>>
      GetAllUserIncomesByUserId(int userId)
      {
            var incomes = await incomeService.GetAllUserIncomes(userId);
            return Ok(incomes);
      }
              …
}
```

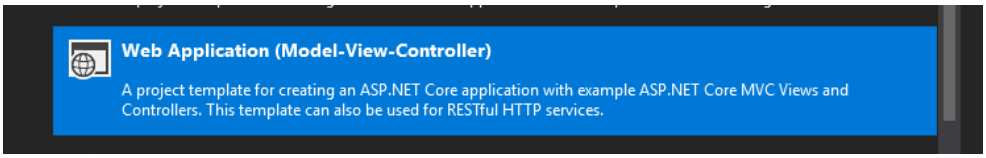
Kôd 3.9. Primjer *IncomeController.cs*

Kôd (Kôd 3.9.) prikazuje primjer *IncomeController.cs* i metode *GetAllUserIncomesByUserId().* U upravitelja je ubrizgano kao ovisnost sučelje *IIncomeService* koje pruža sloj apstrakcije nad servisnim slojem. To predstavlja dobar primjer DIP (engl. *dependecy inversion principle*) principa dizajna koji se odlično slaže s ubrizgavanjem ovisnosti. Cilj DIP principa je izoliranje klasa od njihovih konkretnih implementacija korištenjem apstraktnih klasa ili sučelja. To dovodi do fleksibilnosti u slojevima gdje aplikacija nije nužno vezana uz jednu implementaciju.

Atribut „*Authorize*“ na početku klase označava da je upravitelj autoriziran te da je pristup svim njegovim metodama ograničen samo na autorizirane korisnike. Više govora o autorizaciji bit će u Poglavlju 4.

3.3. Klijent

Raspodijeljeni sustav za kontrolu i vođenje osobnih financija sastoji se od dva klijenta: web i mobilnog klijenta. Mobilni klijent nalazi se u svom zasebnom projektu, dok se web klijent nalazi u API sloju poslužitelja, kao što je navedeno u prijašnjem poglavlju. Tamo je lociran radi činjenice da je u početku stvaranja poslužiteljskog projekta u Visual Studio-u 2019 odabrana opcija *Web Application*, sa slike (Slika 3.8.). Taj predložak (engl. *template*) ne služi samo za razvoj web aplikacija nego i za razvoj RESTful HTTP servisa.



Slika 3.8. Projektni predložak

Tako započeti projekt samo je jedan od načina povezivanja klijenta i poslužitelja u kojemu je iskorišten pogled (engl. *View*) web aplikacije kako bi poslužio klijentsku Vue.js aplikaciju. Vue.js aplikacija je, naravno, mogla biti izdvojena i u svoj zaseban projekt. Tada bi se API projekt kreirao s predloškom *Web API* iz Visual Studio-a. Ovakvim pristupom htio se istražiti drugačiji način inicijalnog postavljanja projekta.

Važno je naglasiti da su oba klijentska projekta razvijana u razvojnom alatu (engl. *Interactive Development Environment*, skraćeno IDE) *Visual Studio Code*.

3.3.1. Web aplikacija

Web klijent predstavlja aplikaciju izvedenu u programskom okviru Vue.js u *Single Page Application* (SPA) tehnologiji.

Vue.js je progresivni JavaScript programski okvir za građenje interaktivnih korisničkih sučelja. Ime programskog okvira Vue fonetska je inačica riječi pogled (engl. *view*) u engleskom jeziku i odgovara pogledu u model-pogled-upravitelj (engl. *Model-View-Controller*, skraćeno MVC) arhitekturi. Nastao je 2014. godine i predstavlja spoj najboljeg dviju svjetova popularnih

JavaScript programskih okvira React i Angular. Najveća prednost u odnosu na ostale JavaScript programske okvire je jako brza krivulja učenja. Prednosti su mu i jako mala veličina paketa, rad s virtualnim DOM-om (engl. *Document Object Model*, skraćeno DOM), reaktivno dvosmjerno povezivanje podataka (engl. *two-way binding*), kompozicija komponenti, jednostavna integracija u već postojeće aplikacije te odlična dokumentacija.

SPA je tehnologija koja omogućuje aplikacijama dohvaćanje sadržaja novootvorene stranice bez učitavanja njezinog HTML-a (engl. *Hypertext Markup Language*, skraćeno HTML), nego se sadržaj te stranice generira dinamički od strane JavaScript-a. HTML se učitava jednom na početku rada aplikacije i nakon toga sadržaj se dinamički mijenja ovisno o tome koju je stranicu korisnik zatražio. To omogućuje korisnicima nastavak rada na stranici dok se ostali elementi u pozadini učitavaju ili osvježavaju. Korisniku se stvara privid puno brže interakcije sa stranicom, a Internet preglednik do tada može dohvaćati podatke s poslužitelja i pripremati neku drugu stranicu. Sadržaj se dinamički dohvaća AJAX (engl. *Asynchronous JavaScript + XML*, skraćeno AJAX) pozivima.

Web projekt generiran je uporabom jednog od mnogih alata iz bogatog ekosustava alata Vue.js-a, Vue CLI (engl. *Command Line Interface*, skraćeno CLI). Vue CLI je sustav za rapidno razvijanje Vue.js aplikacija. Osigurava interaktivno generiranje novih projekata, instantno prototipiziranje komponenti, bogatu kolekciju službenih nadogradnji (engl. *plugins*) i grafičko sučelje za kreiranje i upravljanje Vue.js projektima. Uvelike olakšava rad jer uklanja poteškoće s inicijalnom konfiguracijom projekta, štedi vrijeme prilikom razvoja Vue.js aplikacija te omogućuje fokusiranje na pisanje kôda. Koristeći Vue CLI generirana je Vue.js aplikacija u *ClientApp* direktoriju u API sloju poslužitelja.

Kao jedna od postavki u Vue CLI-u prilikom generiranja aplikacije odabran je i **TypeScript**. TypeScript je nadskup (engl. *superset*) JavaScript-a koji je primarno strogo tipiziran poput objektno-orijentiranih programskih jezika te pruža veću zaštitu programeru od učestalih pogrešaka.

Kako bi se Vue.js aplikacija uspješno pokrenula, nakon pokretanja API projekta, bilo je potrebno konfigurirati oba projekta. U *Startup.cs* klasi API projekta definirano je da se nakon pokretanja projekta standardno (engl. *default*) prikazuje *HomeController.cs* i njegova metoda *Index()* koja prikazuje *Index.cshtml* pogled (Kôd 3.10.).

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-
scale=1.0">
    <title>Expensio</title>
    <link href="~/app.js" rel="preload" as="script">
</head>
<body>
    <div id="app"></div>
    <script type="text/javascript" src="~/app.js"></script>
</body>
</html>
```

Kôd 3.10. *Index.cshtml*

Kako bi se Vue.js aplikacija ispravno poslužila, bilo je potrebno specificirati HTML element * * s atributom *id* i vrijednosti *app*. Navedeni element označava poveznicu s Vue.js aplikacijom koja u *main.ts* datoteci (Kôd 3.11.) specificira postavljanje (engl. *mount*) svog sadržaja u DOM element koji ima atribut *id* s vrijednošću *app*. U *Index.cshtml*-u je potrebno referencirati *app.js* datoteku koju je Vue.js aplikacija generirala kao rezultat kôda koji je napisan u njoj. To se posebno konfiguriralo u Vue.js projektu i njegovoj *vue.config.js* datoteci postavljanjem varijable *outputDir* na *wwwroot* direktorij unutar ishodišnog (engl. *root*) direktorija API projekta. Time je postignuto slanje svih datoteka generiranih od strane Vue.js aplikacije u taj direktorij i one postaju dostupne API projektu za posluživanje. Prikazani način je samo jedna od mogućnosti kako u postojeći ASP.NET Core projekt uključiti Vue.js aplikaciju.

```
new Vue({
render: h => h(App),
router,
store,
components: { App }
```

```
}).$mount('#app')
```

Kôd 3.11. Instanciranje Vue.js aplikacije

U trenutku kada Vue.js klijentska aplikacija želi dohvatiti podatke s API-ja, poziv mora ići putem nekog HTTP klijenta. Kako Vue.js nema svoju vlastitu biblioteku za tu svrhu, u klijentskoj aplikaciji koristila se jedna od popularnijih biblioteka koja to omogućuje - Axios. Axios je HTTP klijent biblioteka za JavaScript temeljena na obećanjima (engl. *Promise*). Korištenje Axios biblioteke omogućuje slanje asinkronih HTTP zahtjeva (engl. *request*) prema API-ju [23].

Axios biblioteka dodana je u klijentsku aplikaciju putem Vue CLI alata pri kreiranju projekta. Da bi biblioteka znala slati zahtjeve (engl. *requests*) na ispravnu adresu poslužitelja, potrebno ju je konfigurirati [24].

```
axios.defaults.baseURL = 'https://localhost:5001/api/';
```

Vrijednost varijable *baseURL* ovisi o okolini (engl. *environment*) u koju je aplikacija postavljena (engl. *deploy*). U produkcijskoj okolini vrijednost ove varijable bit će drugačija jer će i poslužitelj biti na drugoj adresi. Poslužitelj je konfiguriran tako da koristi HTTPS mrežni protokol prilikom komunikacije radi sigurne komunikacije i zaštite osjetljivih podataka. To je detaljno razrađeno u Poglavlju 4.

```
export default class IncomeService {
      public static async getAllUserIncomes(userId: number): Promise<Income[]
      | null>
      {
            try {
                  const response = await axios.get(`income/${userId}/all`);
                  return response.data;
            } catch (error) {
                  Vue.notify({
                  title: 'Error getting incomes',
                  text: error.response.data,
                  type: 'error'
```

```
            });
            return null;
            }
      }
            …
}
```

Kôd 3.12. Primjer TypeScript servisa

Kôd (Kôd 3.12.) je primjer TypeScript servisa koji koristi Axios. Svi servisi u Vue.js aplikaciji su napisani na jednak način. TypeScript klasa *IncomeService.ts* služi kao servisni sloj tj. omotač oko Axios poziva prema poslužitelju. Predočen je primjer metode *getAllUserIncomes()* koja prima kao parametar *userId* što predstavlja *Id* korisnika te vraća polje tipa *Income*. *Income* predstavlja TypeScript reprezentaciju *IncomeDTO* modela vraćenog od strane poslužitelja kao odgovor na klijentov poziv metode, koja se na poslužitelju nalazi mapirana na URL-u '*income/${userId}/all*'. U slučaju iznimke prilikom izvršavanja zahtjeva, podiže se skočna poruka (engl. *toast message*) koja obavještava korisnika o grešci te kao tekst te skočne poruke prikazuje poruku iznimke poslanu s API sloja.
Za prikazivanje podataka pomoću različitih vrsta grafova koristi se biblioteka Vue-chartjs. Vue-chartjs je omot (engl. *wrapper*) oko popularne Chart.js biblioteke. Charts.js je jednostavan i fleksibilan JavaScript programski okvir za prikazivanje podataka u obliku grafova. Vue-chartjs je prilagođen za Vue.js aplikacije kako bi omogućio jednostavnije korištenje biblioteke i stvaranje ponovno iskoristivih graf komponenti.

```
<main-chart
      v-if="loaded"
      chartId="main-chart-01"
      class="chart-wrapper"
      style="height:300px;margin-top:40px;"
      :incomeChartData="groupedUserIncomes"
      :expenseChartData="groupedUserExpenses"
></main-chart>
```

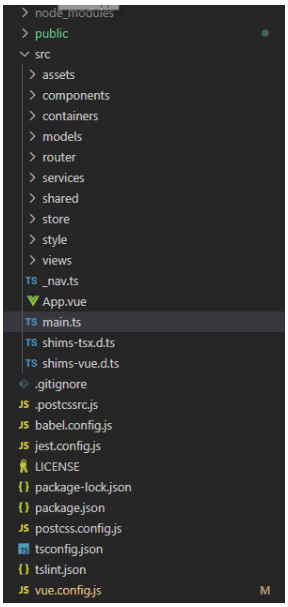
Kôd 3.13. Graf komponenta *main-chart*

U kôdu (Kôd 3.13.) je prikazan primjer graf komponente *main-chart*. Komponente u Vue.js-u neke su od njegovih najmoćnijih funkcionalnosti. Omogućuju enkapsuliranje HTML, CSS i JavaScript kôda u zasebnu cjelinu i ponovnu uporabu na drugom mjestu uz drugačija svojstva. Na apstraktnom nivou, komponente su elementi na koje Vue.js *compiler* dodaje specifična ponašanja [25]. Komponenta *main-chart* prima dva svojstva, *incomeChartData* i *expenseChartData*, s ciljem prikazivanja podataka o prihodima i rashodima. Kao vrijednost tih svojstava postavljena su polja objekata *groupedUserIncomes* i *groupedUserExpenses*. Kreirana su pri stvaranju komponente u metodi životnog ciklusa (engl. *life-cycle hook*) Vue.js komponente *created()* (Kôd 3.14.), dohvaćanjem podataka s API sloja i njihovim grupiranjem po mjesecima. Tako dobiveni podaci spremni su za prikazivanje u grafu.

```
public async created() {
	this.loaded = false;
	let incomes = await IncomeService.getAllUserIncomes(this.userId);
	let expenses = await ExpenseService.getAllUserExpenses(this.userId);
	if (incomes != null && expenses != null) {
		this.userIncomes = incomes;
		this.userExpenses = expenses;
		this.groupedUserExpenses = _(expenses).groupBy((x: Expense) =>
		this.getMonthNameFromDate(x.date)).value();
		this.groupedUserIncomes = _(incomes).groupBy((x: Income) =>
		this.getMonthNameFromDate(x.date)).value();
	}
	this.loaded = true;
}
```

Kôd 3.14. Metoda *created()* u Vue.js aplikaciji

Znak '_' ispred metode *groupBy()* označava JavaScript biblioteku korištenu u aplikaciji – Lodash (Kôd 3.14.). Lodash pruža mnoštvo pomoćnih (engl. *utility*) funkcija za učestale programerske zadatke i omogućuje pisanje konciznijeg i lakše održivog kôda. Konačna struktura Vue.js aplikacije prikazana je na slici (Slika 3.9.).

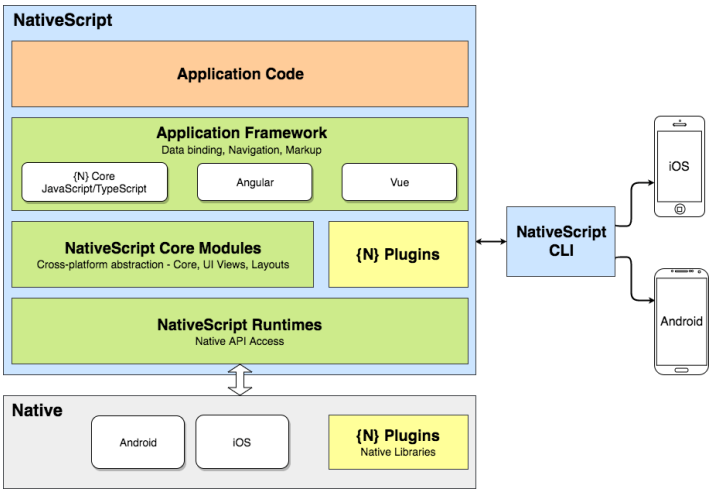


Slika 3.9. Struktura Vue.js aplikacije

3.3.2. Mobilna aplikacija

Nativni razvoj mobilnih aplikacija predstavlja razvoj aplikacija programskim jezicima Java ili Kotlin za Android aplikacije te Objective-C ili Swift za iOS aplikacije. To je najbolji način za razvoj, ali istovremeno uzima najviše vremena jer zahtijeva programiranje iste aplikacije za dvije različite platforme. Također, zahtijeva dobre programerske vještine i znanje razvoja aplikacije na obje platforme. Tu do izražaja dolazi NativeScript [26].

NativeScript-Vue predstavlja programski okvir otvorenog kôda za izgradnju nativnih mobilnih aplikacija uporabom JavaScript-a [27]. NativeScript-Vue je NativeScript nadogradnja (engl. *plugin*) koja omogućuje upotrebu Vue.js-a u razvoju mobilnih aplikacija. Mala veličina paketa i set osnovnih funkcionalnosti čine Vue.js odličnim izborom za razvoj mobilnih aplikacija.



Slika 3.10. Rad NativeScript-a[2]

Slika (Slika 3.10.) prikazuje rad NativeScript-a u pozadini. *NativeScript Runtimes* pretvaraju UI (engl. *user interface*) kôd u Vue.js-u u Native UI kôd za mobilne uređaje. Također, pokreće i JavaScript kôd što omogućuje korištenje JavaScript biblioteka ukoliko nisu napravljene specifično za web aplikacije. NativeScript prevodi sve komponente u nativne komponente za određenu platformu, što znači da će *<ListView></ListView>* komponenta biti *android.widget.ListView* u Android-u te *UITableView* u iOS-u.

NativeScript, također, pruža jednostavno komandno sučelje (skraćeno CLI) koje omogućuje razvoj kôda i njegovo pokretanje s promjenama u realnom-vremenu, koje osvježavaju uređaj/simulator [28].

Globalna NPM (engl. *Node package manager*) instalacija nužna je za korištenje NativeScript CLI-a, a ona se izvršava pozivom naredbe „*npm install – g nativescript*“. Nove aplikacije mogu se kreirati prazne ili koristeći neki od postojećih predložaka. Aplikacijski predlošci pomažu u brzom pokretanju razvoja nativnih višeplatformskih aplikacija koristeći ugrađene UI elemente i najbolje prakse NativeScript-a. Također, skraćuju vrijeme razvoja jer pisanje kôda koji se često ponavlja nije nužno.

U izradi mobilne aplikacije korišten je predložak navigacijska ladica (engl. *Navigation Drawer*). Sadržava *RadSideDrawer* komponentu koja se koristi za navigaciju, a predstavlja izbornik koji se

[2] https://docs.nativescript.org/core-concepts/technical-overview, siječanj 2020.

otvara s jedne strane ekrana mobilne aplikacije putem korisnikovih gesti. Projekt je kreiran kroz komandno sučelje pozivom naredbe „*tns create Expensio –template tns-template-drawer-navigation-vue*“, gdje Expensio predstavlja ime projekta, a ostatak naredbe ime predloška.

RadSideDrawer jedna je od nadogradnji koje postoje za NativeScript aplikacije. NativeScript nadogradnje su NPM paketi s dodanim funkcionalnostima i jednak im je način upotrebe kao NPM paketima za web aplikacije. Na službenoj tržnici nadogradnji (engl. *marketplace*) nalazi se gotovo sve što nije pokriveno osnovnim setom API-ja koje pruža jezgra programskog okvira (npr. nadogradnja za pristup kontaktima mobilnog uređaja i slanje SMS poruka). Tijekom izrade aplikacije korištene su mnoge nadogradnje s tržnice poput *CardView*, *DateTimePicker*, *UI ListView*, *UI Calendar* i *UI Chart*.

NativeScript-Vue aplikacija instancira se drugačije od Vue.js aplikacije. Kada se kreira nova Vue instanca, ne povezuje se na određeni DOM element (Kôd 3.11.) jer NativeScript ne koristi DOM (Kôd 3.15.).

```
new Vue({
  store,
  render: (h) => h(SideDrawer, [h(AuthService.isLoggedIn() ?
routes.Home : routes.Login, { slot: 'mainContent' })])
}).$start();
```

Kôd 3.15. Instanciranje NativeScript-Vue aplikacije

Koristi se funkcija *render()* koja omata (engl. *wrap*) aplikaciju unutar *SideDrawer* komponente. Navigacija postaje uključena kao element koji se prikazuje na svakoj stranici, a sve ostale stranice, koje se prikazuju korisniku, učitavaju se unutar njezinog *slot*-a imena '*mainContent*' (Kôd 3.16.) [29].

```
<RadSideDrawer ref="drawer" drawerLocation="Left"
:gesturesEnabled="gesturesEnabled" @drawerClosed="closeDrawer()">
      <GridLayout  ~drawerContent backgroundColor="#ffffff" columns="*"
      rows="auto, *, auto">
            <Label class="drawer-header" text="Expensio" row="0" col="0"/>
            <StackLayout orientation="vertical" row="1" col="0">
                <Button
                        text="Dashboard"
                        @tap="goToPage(routes.Home)"
```

```
                    class="btn btn-primary"
                ></Button>
                …
            </StackLayout>
                <Button
                    text="Logout"
                    @tap="logout"
                    class="btn btn-primary"
                    row="2"
                    col="0"
                ></Button>
        </GridLayout>
    <Frame ~mainContent>
      <slot name="mainContent"></slot>
    </Frame>
</RadSideDrawer>
```

Kôd 3.16. Prikaz *RadSideDrawer* komponente

Za prikaz podataka u grafičkom obliku u NativeScript-u korištena je nadogradnja *UI Chart*. To je komponenta za grafove dizajnirana specifično za mobilne uređaje. Nudi vrhunske performanse u vremenima učitavanja grafa, funkcionalnosti crtanja te ažuriranja podataka u realnom-vremenu. Postoje dva pogleda koja se uključuju u izgled (engl. *layout*) stranice – *RadCartesianChart* koji se koristi za vizualiziranje podataka u kartezijanskom koordinatnom sustavu te *RadPieChart* koji vizualizira podatke tako da izgledaju poput odsječaka pite (engl. *pie*) [30].

U kôdu (Kôd 3.17.) je prikazan *RadPieChart* za prikaz prihoda.

```
<RadPieChart height="300" allowAnimation="true" row="0">
      <DonutSeries
            v-tkPieSeries
            selectionMode="DataPoint"
            :items="chartData"
            outerRadiusFactor="0.8"
            innerRadiusFactor="0.4"
            expandRadius="0.3"
            valueProperty="amount"
            legendLabel="type"
```

```
            showLabels="true"
        >
        </DonutSeries>
        <RadLegendView
            v-tkPieLegend
            position="Top"
            offsetOrigin="TopRight"
            width="80"
            title="Types"
            enableSelection="true"
        />
</RadPieChart>
```

Kôd 3.17. *RadPieChart* za prikaz prihoda

DonutSeries je tip *ChartSeries*-a koji koristi cirkularni statistički graf podijeljen na odsječke kako bi ilustrirao numeričke proporcije. U prstenastom grafu duljina luka pojedinog odsječka proporcionalna je kvantiteti koju prezentira [31]. Svojstvo *items* predstavlja kolekciju objekata koji predstavljaju vrijednosti za graf, a *valueProperty* određuje ime svojstva u objektu putem kojega će se odrediti proporcije između odsječaka. Svojstvo *legendLabel* određuje koje svojstvo u objektu će poslužiti kao informacija za legendu grafa.

RadLegendView element sadrži informacije o prezentiranim serijama podataka. U kontekstu *RadPieChart*-a, sadrži informacije o različitim prikazanim odsječcima. Svojstvo *position* određuje poziciju legende, a *enableSelection* određuje može li se serija podatkovnih točaka u grafu automatski označiti pritiskom (engl. *tap*) korisnika na stavku legende.

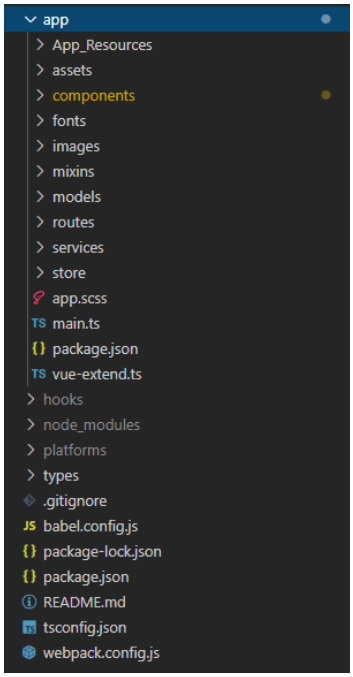
```
public updateChartData(groupedMonthIncomesByType: object) {
        let finalData = [];
        Object.entries(groupedMonthIncomesByType).forEach(([key, val]) =>
        {
            const totalAmount = val
                .map(income => income.amount)
                .reduce((a, b) => a + b, 0);
            finalData.push({ type: key, amount: totalAmount });
        });
```

```
            this.monthlyIncomesChartData = finalData;
    }
```

Kôd 3.18. Metoda *updateChartData()* u *Incomes.vue* komponenti

Kôd (Kôd 3.18.) prikazuje metodu *updateChartData()* u *Incomes.vue* komponenti koja je zadužena za osvježavanje podataka. Ti podaci dalje se prosljeđuju *IncomesChart.vue* komponenti koja sadrži graf prikazan u kôdu (Kôd 3.17.). Metoda prima kao parametar *groupedMonthIncomesByType* objekt koji sadrži korisnikove prihode grupirane po tipu. Tip prihoda označava ključ u objektu, a vrijednost svakog pojedinog tipa je polje tipa *Income.ts* koje predstavlja kolekciju objekata vraćenih s API sloja. Nakon što se zbroje iznosi svih korisničkih prihoda za svaki tip stvara se novi anonimni objekt koji ima dva svojstva, *type* i *amount*. Kreirani objekti spremaju se u polje *finalData* te se dodjeljuju podatkovnom (engl. *data*) svojstvu *monthlyIncomesChartData* koje se dalje proslijeđuje u graf komponentu. Kao što je vidljivo iz kôda (Kôd 3.17.) svojstva *type* i *amount* su svojstva specificirana u grafu i koriste se za određivanje vrijednosti i prikazivanje legende.

Konačna struktura NativeScript-Vue aplikacije prikazana je na slici (Slika 3.11.).



Slika 3.11. Struktura mobilne NativeScript-Vue aplikacije

4. Sigurnost aplikacija

Sigurnost je danas jedna od najbitnijih stavki kod razvoja bilo koje aplikacije ili API-ja koji je javno izložen na Internetu. Osigurava zaštitu resursa na poslužitelju od neovlaštenog pristupa i zlouporabe. Obveza svakog vlasnika informacijskog sustava je maksimalno ga zaštiti i pružiti korisnicima sigurnost pri radu sa sustavom. Posebno se ističe sigurnost u financijskim ili sličnim sustavima koji raspolažu važnim korisničkim podacima poput informacija o kreditnim karticama ili računima koji čuvaju neku vrstu vrijednosti (npr. dionice ili kripto valute). Kako bi se očuvala sigurnost informacijskih sustava, svakom korisniku moraju biti dodijeljene vlastite vjerodajnice te uloge koje definiraju kojem dijelu sustava ima pravo pristupiti i što može činiti. Ti procesi nazivaju se autentikacija i autorizacija. Autentikacija je proces u kojem korisnik šalje svoje vjerodajnice na provjeru. One se, zatim, uspoređuju s onima koje su za njega već pohranjene na operacijskom sustavu, u bazi podataka ili aplikaciji. Ako se podudaraju, smatra se da se korisnik uspješno autenticirao u sustav i može izvršiti akcije za koje je autoriziran u procesu autorizacije. Autorizacija podrazumijeva proces koji određuje što korisnik smije napraviti u sustavu [32].

Jedna od stavki koja uvelike doprinosi sigurnosti informacijskih sustava je i korištenje HTTPS mrežnog protokola. HTTPS je mrežni protokol koji osigurava komunikaciju između dva sustava. HTTPS radi na istim principima kao HTTP mrežni protokol, ali je puno sigurniji jer je prijenos informacija enkriptiran. Enkriptirani podaci onemogućuju hakerima ili drugim zlonamjernim stranama čitanje i modifikaciju sadržaj podataka prilikom prijenosa između dva sustava [33].

Raspodijeljeni sustav za kontrolu i vođenje osobnih financija osiguran je HTTPS mrežnim protokolom. Sav promet između klijenata i poslužitelja osiguran je i enkriptiran. Da bi se poslužitelj konfigurirao i koristio HTTPS bilo je potrebno dodati sljedeći kôd u *Startup.cs* klasu API sloja:

```
app.UseHttpsRedirection();
```

4.1. Microsoft ASP.NET Core Identity

Primarni mehanizam u ASP.NET Core-u za identificiranje aplikacijskih korisnika je ASP.NET Core Identity članski sustav (engl. *membership system*). ASP.NET Core Identity sprema korisničke informacije poput korisničkog imena, lozinke, uloga i tvrdnji (engl. *claims*) u skladište podataka (engl. *user store*) konfigurirano od strane programera. Tipično ASP.NET Core Identity sprema podatke u SQL Server bazu podataka putem Entity Framework-a. Podatke se može spremati i u vlastita rješenja za pohranu podataka ili rješenja trećih strana (engl. *third party*) poput *Azure Table Storage*-a, *CosmosDB*-a ili nečeg drugog [34]. Osim što pruža opcije za skladištenje podataka, ASP.NET Core Identity je također skup API-ja i pomoćnih funkcija koje uvelike olakšavaju rad u procesu upravljanja korisnicima.

U početku razvoja poslužiteljskog projekta nije odabran autentikacijski predložak koji uključuje prekonfiguriran ASP.NET Core Identity. Identity je dodan kao zaseban element što je omogućilo veću konfiguraciju skladišta korisničkih podataka te entitetskih klasa koje će biti modeli za te podatke. Za skladištenje podataka odabrana je SQL Server baza podataka uz Entity Framework Core kao ORM iz razloga što se ista kombinacija koristi i za kreiranje svih drugih tablica u sustavu te skladištenje podataka.

Najprije je bilo potrebno napraviti instalaciju NuGet biblioteke *Microsoft.AspNetCore.Identity.EntityFrameworkCore* prikazanu slikom (Slika 4.1.).

Slika 4.1. Instalacija *Microsoft.AspNetCore.Identity.EntityFrameworkCore*-a

Ta biblioteka osigurati će integraciju ASP.NET Core Identity-a u postojeći projekt koji koristi EF Core.

Identity model predstavlja model korisničkog entiteta u bazi podataka i definiran je sljedećim entitetima: *User*, *Role*, *UserClaim*, *UserToken*, *UserLogin*, *RoleClaim* i *UserRole*. Najvažniji od

njih su *User* i *Role* jer predstavljaju model korisnika i ulogu (engl. *role*) koju korisnik ima u sustavu.

ASP.NET Core Identity omogućuje dodatne konfiguracije i proširivanja svojih osnovnih modela. Potrebno je samo da nova entitetska klasa naslijedi odgovarajuću klasu iz Identity-a. Kôd (Kôd 4.1.) prikazuje izgled *IdentityUser<TKey>* klase koja je bazna klasa Identity-a, a predstavlja korisnika. *TKey* ostavlja programeru mogućnost definiranja tipa podatka koji će se koristiti za primarni ključ tablice koja će nastati iz te entitetske klase.

```
public class IdentityUser<TKey> where TKey : IEquatable<TKey>
{
    …
    public IdentityUser(string userName);
    public virtual string SecurityStamp { get; set; }
    public virtual bool PhoneNumberConfirmed { get; set; }
    public virtual string PhoneNumber { get; set; }
    public virtual string PasswordHash { get; set; }
    public virtual string NormalizedUserName { get; set; }
    public virtual string NormalizedEmail { get; set; }
    public virtual DateTimeOffset? LockoutEnd { get; set; }
    public virtual bool LockoutEnabled { get; set; }
    public virtual TKey Id { get; set; }
    public virtual bool EmailConfirmed { get; set; }
    public virtual string Email { get; set; }
    public virtual string ConcurrencyStamp { get; set; }
    public virtual int AccessFailedCount { get; set; }
    public virtual bool TwoFactorEnabled { get; set; }
    public virtual string UserName { get; set; }
    public override string ToString();
}
```

Kôd 4.1. Prikaz bazne klase *IdentityUser<TKey>*

U ovom radu napravljena je nova klasa *ApplicationUser* koja predstavlja korisnika. U odnosu na baznu klasu koju nasljeđuje proširena je dodatnim svojstvima potrebnim za rad sustava. Također, kao što je vidljivo u kôdu (Kôd 4.2.), promijenjen je primarni ključ u tip podatka *int* (engl. *integer*).

```
public class ApplicationUser : IdentityUser<int>
      {
            public string FirstName { get; set; }
            public string LastName { get; set; }
            public DateTime DateOfBirth { get; set; }
            public DateTime RegistrationDate { get; set; }
            public string Country { get; set; }
            public string Gender { get; set; }
            public bool IsChildAccount { get; set; }
            public int? AccountId { get; set; }
            public virtual ICollection<Income> Incomes { get; set; }
            public virtual ICollection<Expense> Expenses { get; set; }
            public virtual ICollection<AccountLimits> AccountLimits { get;
            set; }
      }
```

Kôd 4.2. Prikaz *ApplicationUser* klase

Isti princip primijenjen je i za klasu *ApplicationRole* koja je naslijedila *IdentityRole*<*int*>, promijenila joj primarni ključ te dodala samo jedno svojstvo – Opis (engl. *Description*).

Kako bi se proširenja Identity modela reflektirala u bazi podataka, potrebno je konfigurirati *IdentityDbContext* klasu, baznu klasu Microsoft Identity-a za uporabu s EF Core-om. Također, potrebno je konfigurirati i EF Core kako bi radio s ASP.NET Core Identity kontekstom. Prikaz klase *ExpensioContext* nakon svih konfiguracija može se vidjeti u kôdu (Kôd 3.4.). Ona sada u pozadini nasljeđuje *IdentityDbContext* klasu. Konfigurirana je tako da koristi prilagođene klase *ApplicationUser* i *ApplicationRole* te da primarni ključ tablica Microsoft Identity-a bude *int*, a ne *string* kako je bilo postavljeno nakon instalacije paketa.

Nakon konfiguracije ASP.NET Core Identity-a potrebno ga je registrirati kao servis u aplikaciju. To se radi u metodi *ConfigureServices()* u klasi *Startup.cs* pozivom metode *AddIdentity()*. Uz *AddIdentity()* metodu poziva se metoda *AddEntityFrameworkStores()* koja registrira *ExpensioContext* implementaciju Identity-a.

```
services.AddIdentity<ApplicationUser, ApplicationRole>(options =>
{
      options.User.RequireUniqueEmail = true;
}).AddEntityFrameworkStores<ExpensioContext>();
```

4.2. JSON Web Token

JSON Web Token (skraćeno JWT) je standard koji definira kompaktan način sigurne autentikacije web aplikacije u obliku JSON objekta. JWT protokoli za provjeru autentičnosti i autorizacije koriste token kao metodu prijenosa podataka između klijenta i poslužitelja kako bi se u konačnici korisnik ili ovlastio za izvršavanje radnje ili zatražio podatke iz nekog resursa. JWT se koristi za slanje informacija koje mogu biti verificirane i provjerene s digitalnim potpisom. Sadrži kompaktni JSON objekt koji je kriptografski potpisan da bi mu se verificirala autentičnost. Može biti i enkriptiran ukoliko objekt sadrži povjerljive informacije [35].

JSON Web Token sastoji se od tri dijela: zaglavlja (engl. *Header*), tereta (engl. *Payload*) i potpisa (engl. *Signature*) odijeljenih točkom. Izgled tokena je sljedeći:

```
ZAGLAVLJE.TERET.POTPIS
```

Zaglavlje se tipično sastoji od dva dijela: tipa tokena i *hash* algoritma koji je upotrijebljen za potpis tokena. Teret sadrži informacije o korisniku u obliku tvrdnji. I zaglavlje i teret tokena su enkodirani u Base64Url shemi. Potpis, kao posljednji dio, potvrđuje da se token nije mijenjao na putu. Funkcionira tako da spaja enkodirano zaglavlje i teret i potpisuje ih koristeći jaki enkripcijski algoritam naveden u zaglavlju tokena, a koristeći sigurnosni ključ definiran na poslužitelju koji ih izdaje. Time se omogućuje da samo poslužitelj ima ključ i da može verificirati postojeće ili izdavati nove tokene. Treba naglasiti da potpis ne osigurava enkripciju podataka u dijelu tereta. Oni su i dalje u Base64Url obliku što znači da su vidljivi svima. Tajni podaci se ne bi trebali zapisivati u teret ukoliko nisu posebno enkriptirani.

Podrška za JWT tokene ugrađena je u ASP.NET Core te je potrebno konfigurirati autentikacijski *middleware* kako bi koristio JWT tokene za autentikaciju, a ne neki drugi način poput kolačića (engl. *cookies*). Konfiguracija se odrađuje u metodi *ConfigureServices()* u klasi *Startup.cs*. (Kôd 4.3.)

```
var tokenConfig =
Configuration.GetSection("TokenManagement").Get<TokenManagement>();
services.AddAuthentication(options =>
{
options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
      options.TokenValidationParameters = new TokenValidationParameters
      {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = tokenConfig.Issuer,
            ValidAudience = tokenConfig.Audience,
            IssuerSigningKey = new
            SymmetricSecurityKey(Encoding.UTF8.GetBytes(tokenConfig.Secret))
      };
});
```

Kôd 4.3. Konfiguracija JWT tokena

Da bi se konfigurirala autentikacija, potrebno je dohvatiti postavke koje će se koristiti tijekom konfiguracije. Postavke se nalaze u *appsettings.Development.json* (Kôd 4.4.) datoteci, a mapiraju se na POCO (engl. *plain old CLR object*) klasu *TokenManagement.cs* (Kôd 4.5.).

```
"TokenManagement": {
    "Secret": "********************",
    "Issuer": "ExpensioApi",
    "Audience": "ExpensioFrontend",
    "AccessExpiration": 1
  }
```

Kôd 4.4. Konfiguracijske postavke za *TokenManagement*

```
public class TokenManagement
{
    public string Secret { get; set; }
    public string Issuer { get; set; }
    public string Audience { get; set; }
    public int AccessExpiration { get; set; }
}
```

Kôd 4.5. *TokenManagement* POCO klasa

Konfiguracija JWT tokena odvila se u *AddJwtBearer()* metodi kreiranjem novog objekta tipa *TokenValidationParameters*. U njemu se specificira koje se provjere žele odraditi prilikom verifikacije tokena i što će se koristiti kao vrijednost za usporedbu u provjerama. U ovom dijelu dolazi do izražaja *tokenConfig* koji je dohvatio konfiguracijske postavke jer se upravo one koriste kako bi se verificirao token. Svojstvo objekta *IssuerSigningKey* definira koji će se ključ koristiti pri usporedbi potpisa tokena. Kako bi verifikacija uspješno prošla, to mora biti isti ključ koji je korišten za potpis tokena prije kreiranju.

Nakon što se autentikacijski *middleware* konfigurirao za korištenje JWT-a, potrebno je uključiti korištenje autentikacije u sustavu jer inicijalno nije postavljeno. Uključuje se pozivom metode *UseAuthentication()* u metodi *Configure()* u *Startup.cs* klasi. Ta metoda dodaje prethodno konfigurirani autentikacijski *middleware* u aplikacijski tijek izvršavanja kako bi svaki zahtjev prema poslužitelju prošao kroz autentikacijski *middleware* i bio verificiran.

Dodavanjem autentikacije u aplikacijski tijek izvršavanja omogućeno je postavljanje atributa *[Authorize]* nad upraviteljem. To osigurava primjenu JWT autentikacije na svakom pozivu koji ide prema upravitelju, a da na sebi ima *[Authorize]* atribut. Svi upravitelji unutar poslužiteljskog projekta imaju taj atribut postavljen osim *AuthenticationController.cs* upravitelja. On nema postavljen niti jedan atribut što znači da mu se ne zabranjuje pristup. Dodatno sve metode unutar tog upravitelja imaju postavljene posebne atribute *[AllowAnonymous]* kako bi im se moglo pristupiti prilikom registracije korisnika ili prilikom prijavljivanja postojećeg korisnika u sustav. U tim trenucima korisnik još nema izdan token koji može koristiti te iz tog razloga nema smisla niti provjeravati isti prilikom pristupa tim metodama (Kôd 4.6.).

```
[AllowAnonymous]
[HttpPost("login")]
public async Task<ActionResult<UserDto>> RequestToken([FromBody] LoginRequest
request)
{
      var user = await userService.AuthenticateUser(request);
      if (user == null) return Unauthorized("Invalid credentials. Please try
      again.");
      var userRoles = await userService.GetUserRoles(user.Id);
      if (userRoles.Any()) user.Role = userRoles[0];
      var token = userService.CreateToken(user, settings.TokenManagement);
      user.Token = token;
      return Ok(user);
}
```

Kôd 4.6. *RequestToken()* pristupna točka

Izdavanje tokena odvija se u metodi *CreateToken()* u *UserService.cs* servisu (Kôd 4.7.). Metoda kao parametre prima korisnika koji je uspješno verificiran u sustavu te konfiguracijske postavke za tokene.

```
public string CreateToken(UserDto user, TokenManagement tokenSettings)
{
      var claims = new[]{
            new Claim(CustomClaimTypes.Username, user.UserName),
            new Claim(CustomClaimTypes.FirstName, user.FirstName),
            new Claim(CustomClaimTypes.LastName, user.LastName),
            new Claim(CustomClaimTypes.Role, user.Role),
            new Claim(CustomClaimTypes.UserId, user.Id.ToString()),
            new Claim(CustomClaimTypes.AccountId, user.AccountId.ToString())
      };
      var key = new
      SymmetricSecurityKey(Encoding.UTF8.GetBytes(tokenSettings.Secret));
      var credentials = new
      SigningCredentials(key,SecurityAlgorithms.HmacSha256);
      var token = new JwtSecurityToken(
```

```
            issuer: tokenSettings.Issuer,
            audience: tokenSettings.Audience,
            claims: claims,
            expires: DateTime.Now.AddDays(tokenSettings.AccessExpiration),
             signingCredentials: credentials);
             var generatedToken = new
            JwtSecurityTokenHandler().WriteToken(token);
            return generatedToken;
}
```

Kôd 4.7. *CreateToken()* metoda

U toj metodi postavljaju se tvrdnje o korisniku koje se žele zapisati u token kao teret. Nakon toga potrebno je generirati potpis tokena stvaranjem novog objekta tipa *SigningCredentials* predajući mu kao parametre korišteni algoritam za potpis te ključ definiran u konfiguracijskim postavkama tokena. Taj ključ mora odgovarati ključu koji se koristi za verifikaciju tokena. Nakon što se token uspješno generira, vraća se klijentu kako bi ga on mogao uključiti u svaki svoj sljedeći zahtjev prema poslužitelju. Klijent to radi na način da u autorizacijsko zaglavlje svakog zahtjeva uključi token koji je dobio s poslužitelja u formatu „Bearer *token*“ (Kôd 4.8.). To će osigurati ispravnu verifikaciju klijentskih zahtjeva od strane poslužitelja te vraćanje odgovora koji sadrži tražene resurse.

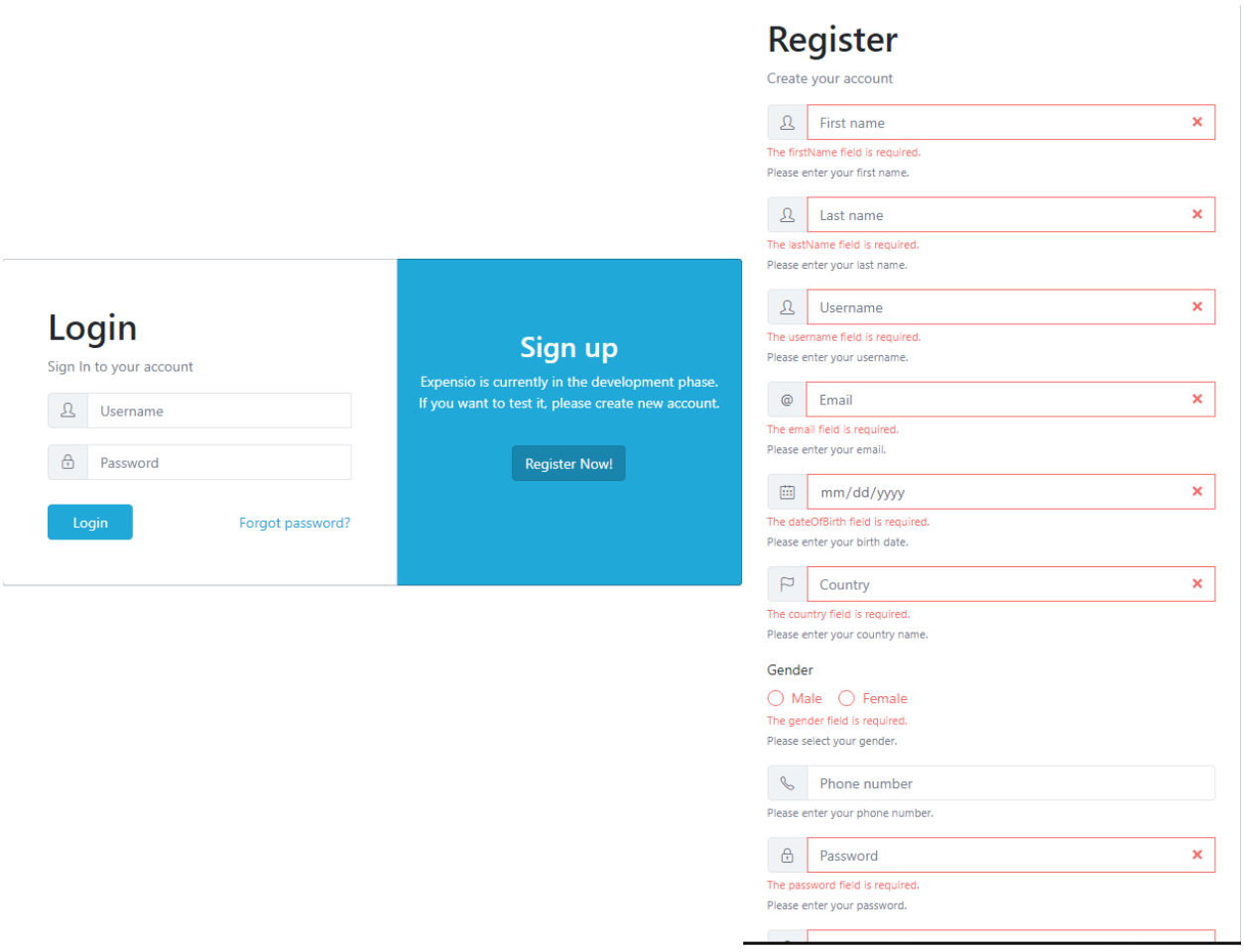
```
axios.interceptors.request.use(
  (config) => {
    let token = localStorage.getItem('authToken');
    if (token) {
      config.headers['Authorization'] = `Bearer ${ token }`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);
```

Kôd 4.8. Dodavanje tokena u zaglavlje zahtjeva

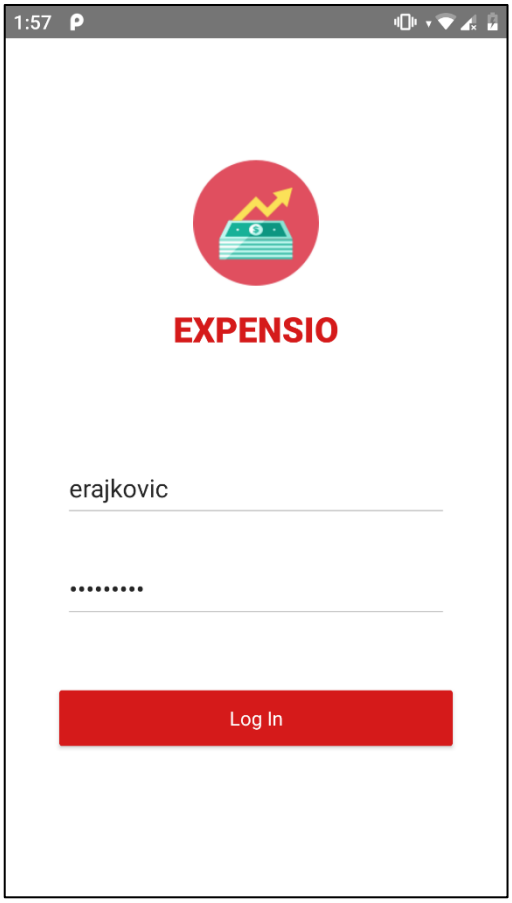
5. Korištenje aplikacija

Opis korištenja aplikacije popraćen je prikazima ekrana web i mobilne aplikacije za pojedine značajke.

Korisnik pristupa aplikaciji ispunjavajući formu za prijavu (engl. *login form*), unošenjem korisničkog imena i zaporke (Slika 5.1. i Slika 5.2.). U slučaju prve prijave u sustav najprije je potrebno ispuniti registracijsku formu. Forma za prijavu i registracijska forma imaju postavljene validacije obaveznih polja.

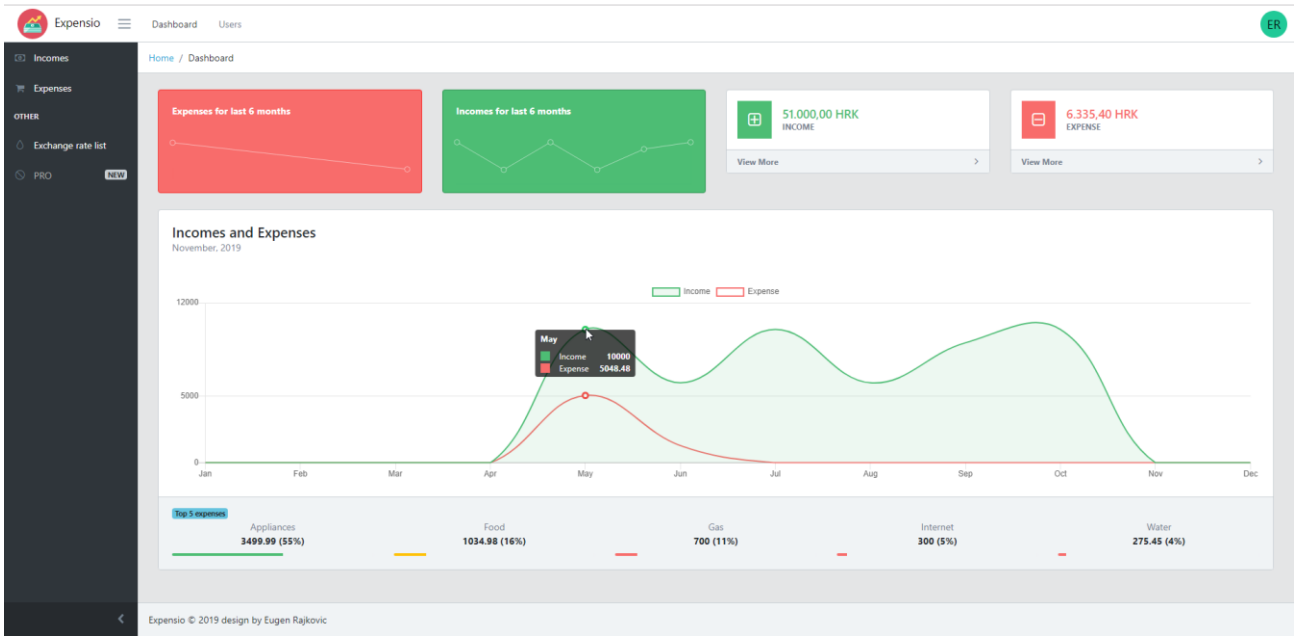


Slika 5.1. Registracijska i *login* forma za pristup web aplikaciji

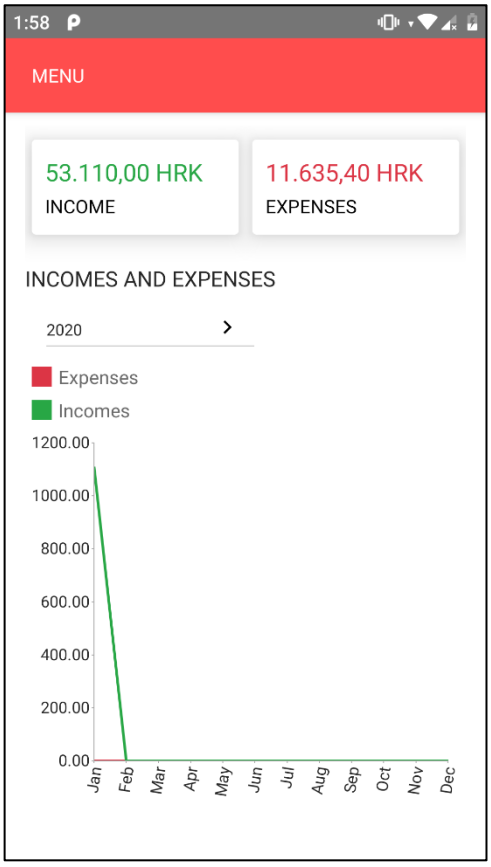


Slika 5.2. Ekran za prijavu u mobilnu aplikaciju

Na početnoj stranici, prikazanoj na slikama (Slika 5.3.) za web i (Slika 5.4.) za mobilnu aplikaciju, može se vidjeti sveobuhvatni linijski grafikon koji prikazuje informacije kao niz podatkovnih točaka koji čine dvije linije: prihodi (engl. *incomes*) i rashodi (engl. *expenses*). X os dijagrama čini popis mjeseci u godini, a y os predstavlja određeni novčani iznos. Time je omogućen usporedni prikaz svih prihoda i rashoda u određenom vremenskom periodu. U donjem dijelu grafa različitim bojama označeni su udijeli pet najčešćih tipova troškova kako bi korisnik odmah dobio slikovit uvid u vlastite rashode. Na stranici se nalaze i razdvojeni linijski dijagrami prihoda i rashoda kako bi se omogućilo praćenje „rasta“ ili „pada“ istih. Također, kartično su predočeni iznosi cjelokupnih prihoda ili rashoda. Pritiskom na neku od kartica korisnika se preusmjerava na istoimenu stranicu (*Incomes* ili *Expenses*) na kojoj se nalazi detaljna razrada.

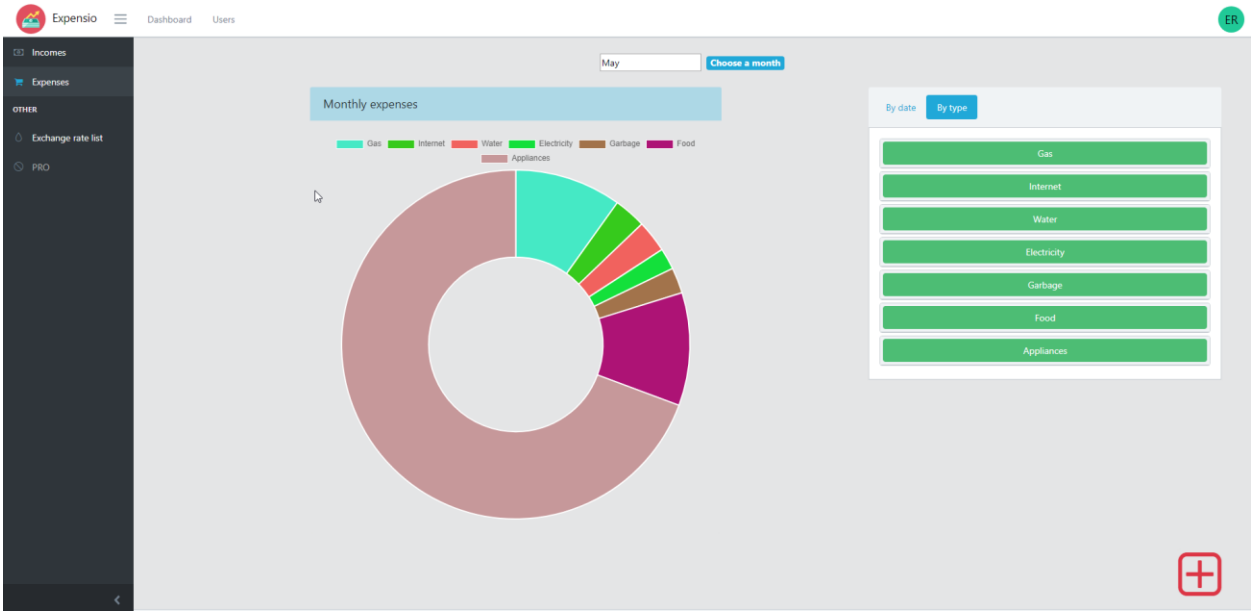


Slika 5.3. Početna stranica web aplikacije

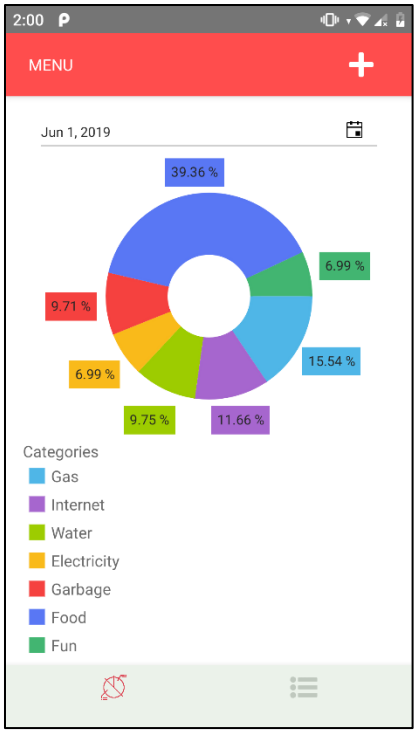


Slika 5.4. Početna stranica mobilne aplikacije

Po dolasku na stranicu *Expenses* najprije je potrebno odabrati mjesec kako bi korisnik dobio željene podatke. Nakon prikaza mjeseca, dobivaju se informacije o troškovima podijeljenim po kategorijama (npr. prijevoz, režije, hrana, uređaji itd.) ili prema datumu kada je trošak izvršen. Sve to prikazano je u obliku kružnog grafikona (engl. *doughnut chart*) i predočeno slikama (Slika 5.5.) za web i (Slika 5.6.) za mobilnu aplikaciju.

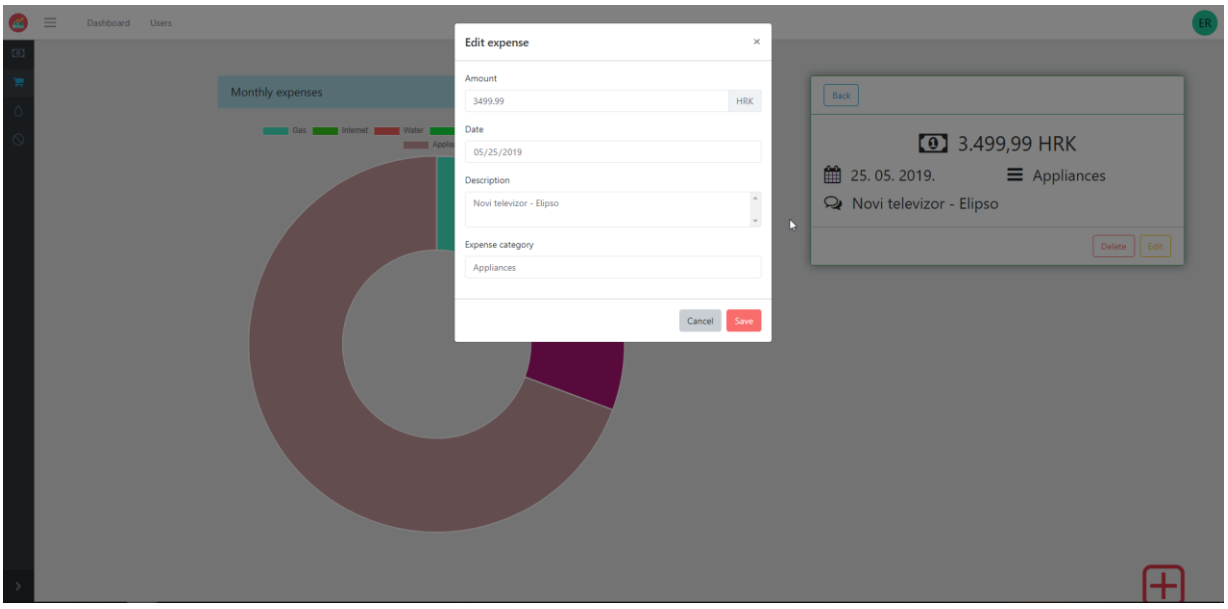


Slika 5.5. Prikaz stranice *Expenses* na web aplikaciji

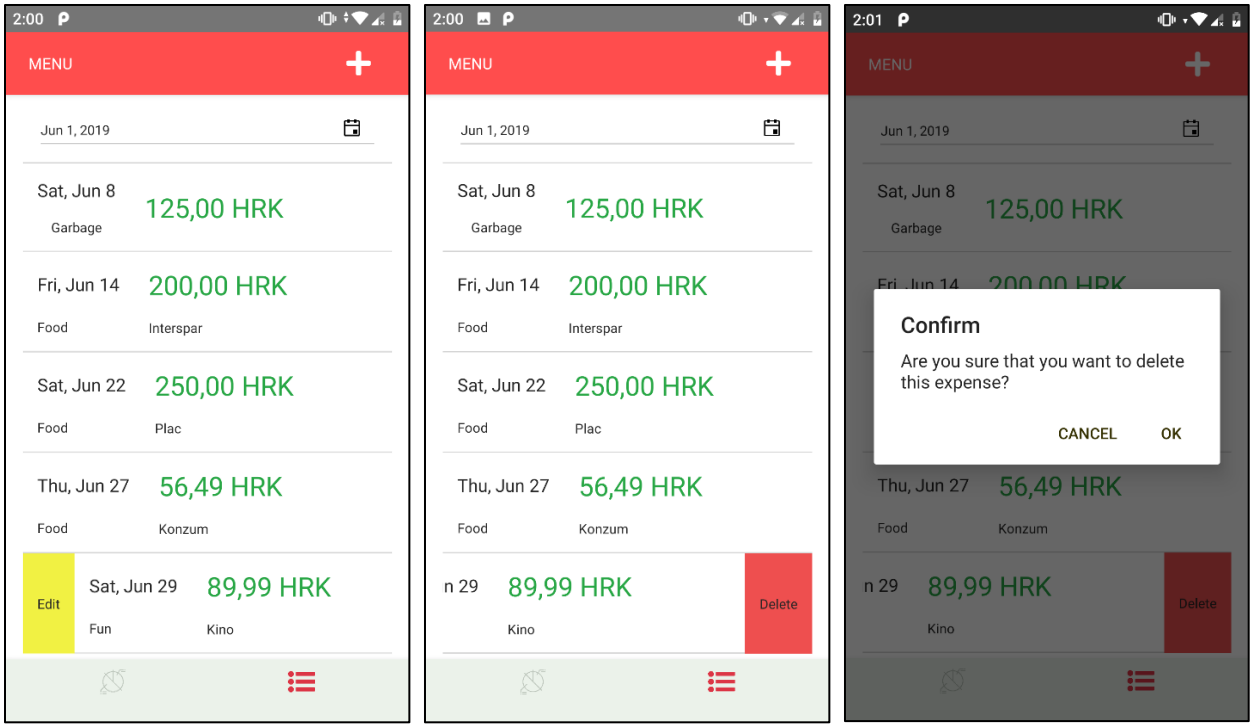


Slika 5.6. Prikaz stranice *Expenses* na mobilnoj aplikaciji

Svaki rashod moguće je detaljno tekstualno opisati. Rashode je moguće retrogradno ažurirati, kao i brisati postojeće te dodavati nove troškove. To prate promjene grafikona u realnom vremenu, što je prikazano slikama (Slika 5.7.) za web i (Slika 5.8.) za mobilnu aplikaciju.



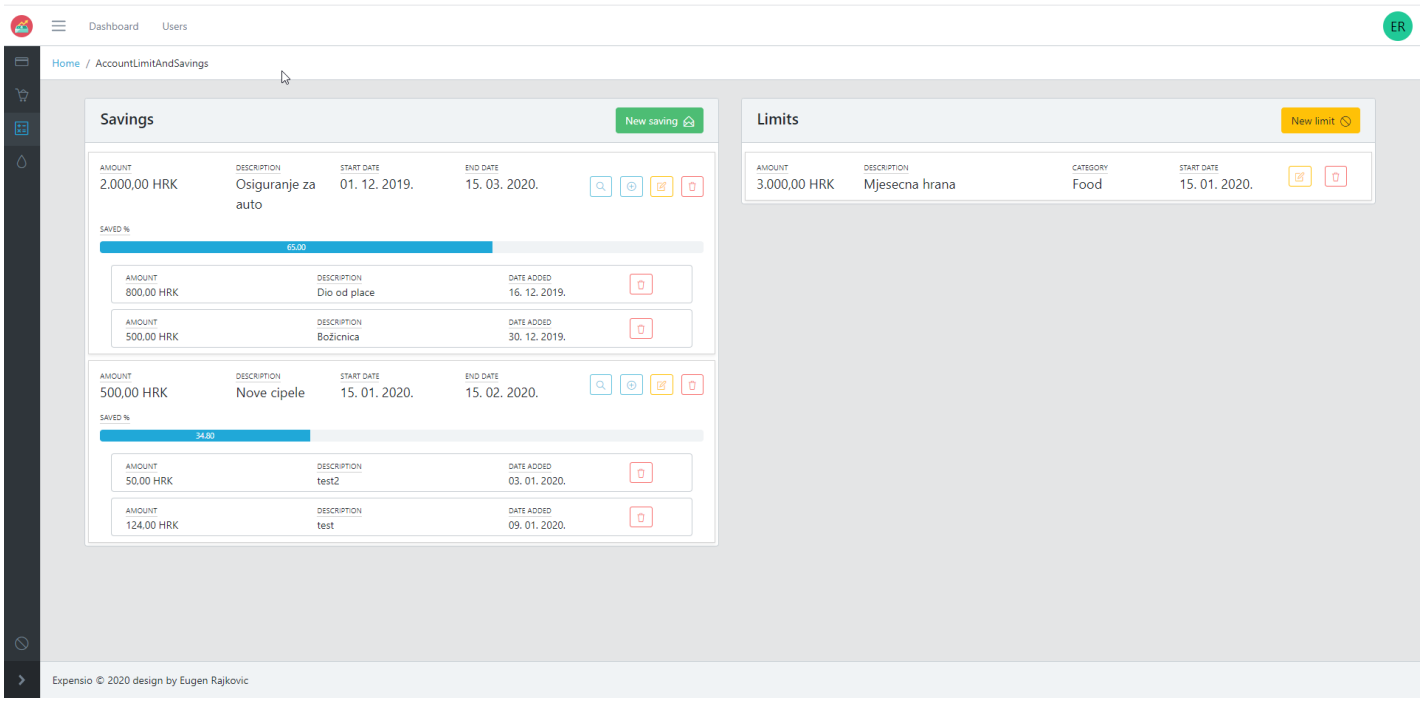
Slika 5.7. Ažuriranje pri unosu podataka



Slika 5.8. Opcije nad pojedinim stavkama troškova

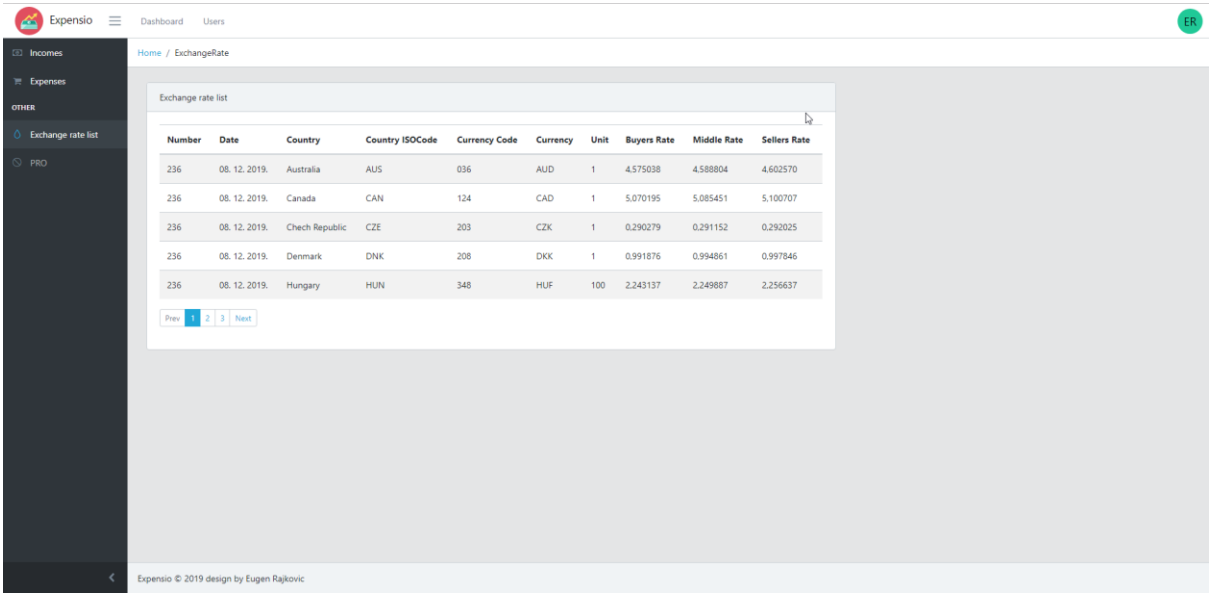
Slično sučelje čini stranicu *Incomes* uz primjenu odgovarajućih kategorija prihoda.

Na stranici *Savings and limits* nalaze se odvojene istoimene kartice koje se sastoje od harmonika (engl. *accordion*). Predstavljaju sustav omotnica (engl. *envelope system*) kojim korisnik može pratiti koliko točno ima novaca u svakoj kategoriji vlastite štednje. Na sličan način, korisnik može postaviti određena ograničenja (engl. *limits*) u potrošnji na određenu kategoriju i za određeni vremenski period. Svaka harmonika predstavljena je još jednom karticom koja sadrži glavne podatke o štednji (engl. *saving*) ili ograničenju (engl. *limit*). Ispod kartice nalazi se traka napretka (engl. *progress bar*). Njome je omogućeno praćenje napretka u štednji ili ograničenju, izraženog postotkom ostvarenog željenog iznosa u trenutku pregledavanja. Pritiskom na harmoniku štednje otvaraju se njezini detalji. Prikazuju svaku uplatu koju je korisnik napravio u konkretnu štednu „omotnicu“ (Slika 5.9.). Svako ograničenje ili štednju je moguće uređivati i brisati.

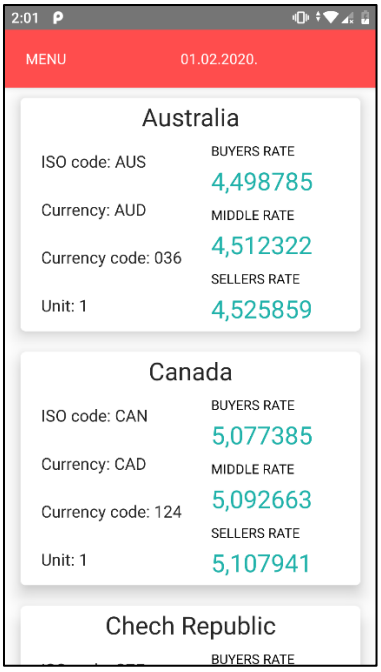


Slika 5.9. Stranica *Savings and limits* web aplikacije

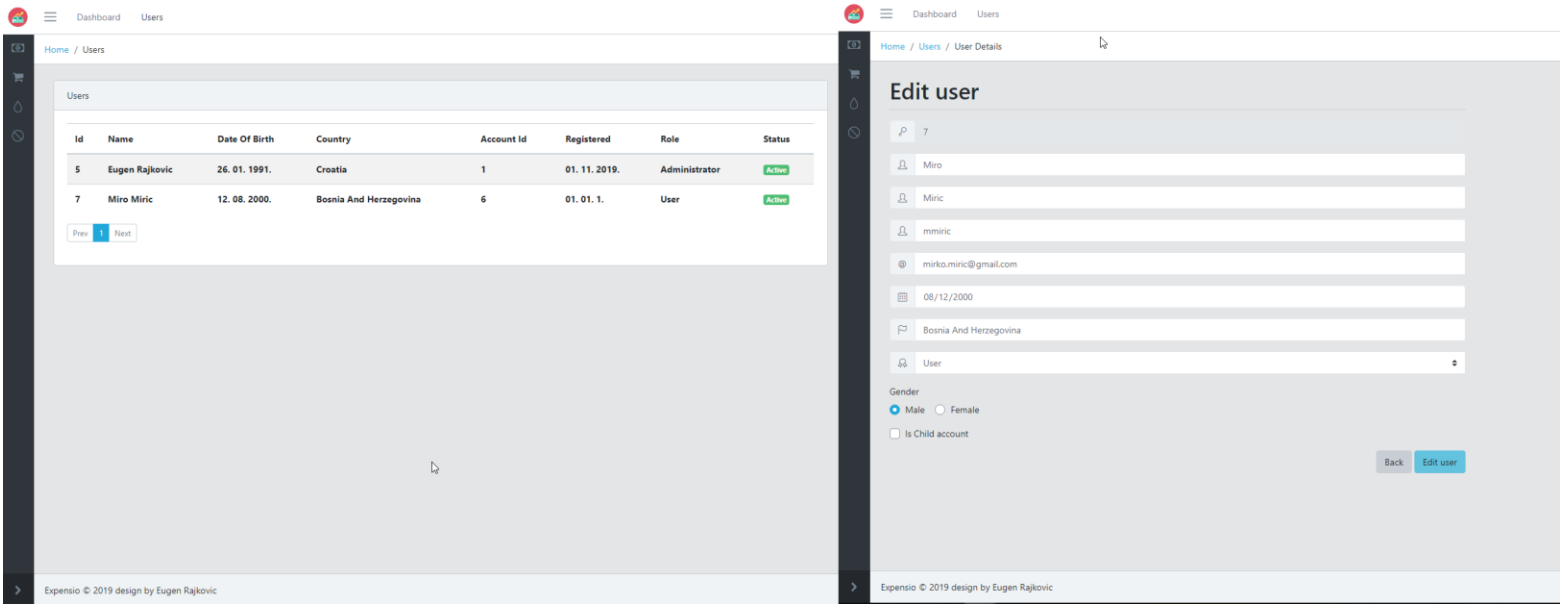
Stranica *Exchange rate list*, predočena slikama (Slika 5.10.) za web i (Slika 5.11.) za mobilnu aplikaciju, prikazuje podatke aktualne tečajne liste Hrvatske narodne banke koja se svakodnevno ažurira.



Slika 5.10. Tečajna lista HNB-a u web aplikaciji

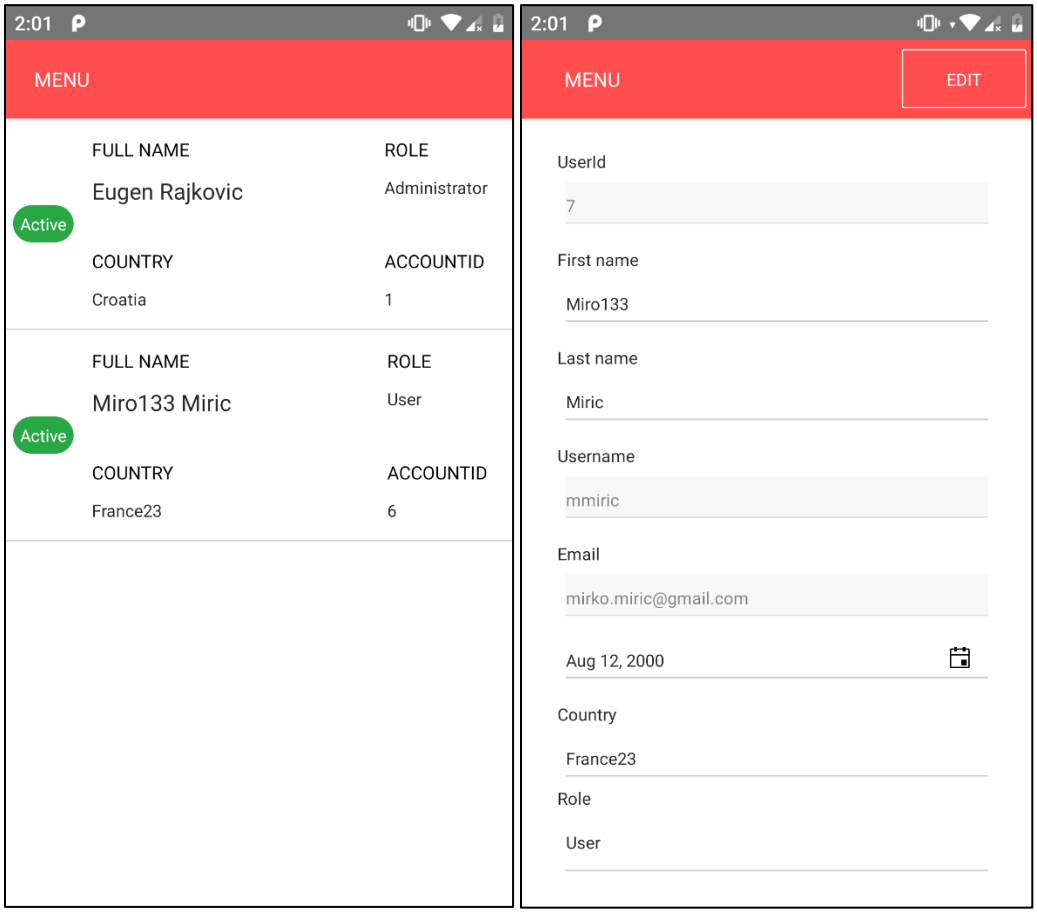


Slika 5.11. Tečajna lista HNB-a u mobilnoj aplikaciji



Slika 5.12. Administracijska ploča web aplikacije

Na slikama (Slika 5.12.) za web i (Slika 5.13.) za mobilnu aplikaciju prikazane su administracijske ploče gdje se na početnoj stranici ispisuje popis korisnika s pripadajućim osnovnim informacijama i statusom u sustavu, a na stranici s korisničkim detaljima (engl. *User Details*) pristupa se svim informacijama o korisniku koje je moguće ažurirati. Pristup ovim stranicama moguć je samo administratorskim korisnicima.



Slika 5.13. Administracijska ploča mobilne aplikacije

6. Analiza aplikacija

Značajke aplikacije analizirane su u odnosu na postojeća rješenja. Također, testna faza u krugu prijatelja i obitelji pomogla je u evaluaciji korisničkog doživljaja aplikacije i ukazala na potencijalne preinake i nadogradnje aplikacije.

6.1. Usporedba s postojećim rješenjima

Aplikacija je uspoređena s alatima opisanim u drugom poglavlju: *Mint, You Need a Budget, Wally* i *Toshl Finance*. Kako bi se učinila što kvalitetnija analiza, usporedba je izvršena po idućim kategorijama: dostupnost za web/mobitel, naknada za uporabu, mogućnost povezivanja s bankovnim računom, georestrikcija, kategorizacija financija i grafički prikaz. Usporedba je prikazana u tablici (Tablica 6.1.), gdje simbol + predstavlja prisutnost određene značajke, a – odsutnost.

Tablica 6.1. Usporedba razvijenog sustava s postojećim rješenjima

	Mint	**YNAB**	**Wally**	**Toshl**	**Razvijeni sustav**
dostupnost za web/mob	+/+	+/-	+/+	+/+	+/+
naknada za uporabu	-	+	-	- **postoji samo u Pro verziji*	-
veza s bankovnim računom	+	- **postoji samo u naprednoj verziji*	-	+	-
georestrikcija	+	+ **na određene značajke*	+	-	-
kategorizacija financija	+	+	+	+	+
grafički prikaz	+	+	+	+	+

U kategorijama dostupnost na webu/mobitelu, grafički prikaz i kategorizacija financija sve analizirane aplikacije sadrže navedene značajke s različitim idejnim rješenjima. Ipak, razlike postoje u kategoriji georestrikcije. Pokazalo se da su Toshl i razvijeni sustav sasvim neovisni o gelokaciji, dok kod ostalih postoji barem neka vrsta restrikcije značajki ovisno o mjestu uporabe. Vezu s bankovnim računom nudi aplikacija Mint, dok YNAB i Toshl samo u naprednijim verzijama. Povezivanje s bankovnim sustavima u Hrvatskoj nije moguće ostvariti kao privatna osoba, stoga razvijeni sustav ne sadrži mogućnost takve sinkronizacije. Naknadu za uporabu zahtjeva YNAB i Toshl u naprednoj verziji, dok je korištenje Mint-a, Wally-a i razvijenog sustava potpuno besplatno.

6.2. Moguća proširenja aplikacija

U testnoj fazi aplikacija je dana obitelji i prijateljima na korištenje. Jedanaest korisnika aktivno je koristilo aplikaciju mjesec dana i unosilo podatke o vlastitim financijama. Došlo se do zaključka da su svi korisnici subjektivno imali bolji uvid u vlastite financijske obrasce i osobne navike. To je nerijetko utjecalo na donošenje drugačijih financijskih odluka u odnosu na razdoblje bez uporabe alata za praćenje financija. Shodno tome korisnici navode da su u testnom periodu najviše uštedjeli u kategorijama zabava/izlasci, namirnice i prijevoz. Aplikacija im je, s tehničke strane, bila jednostavna za korištenje. Budući da postoji web i mobilna verzija aplikacije, različite potrebe korisnika su ispunjene te njezina uporaba nije zahtijevala specifično mjesto i vrijeme. Smatrali su da je grafički prikaz ono što im je najviše pomoglo u boljoj percepciji vlastitih prihoda i rashoda. Kategorizacija troškova pokazala se kao segment koji im je najviše nedostajao u prijašnjem vođenju financija. Na temelju povratnih informacija i sugestija zamišljena su i skicirana moguća proširenja aplikacije:

- Unos trajnih naloga koji bi u zadanim vremenskim intervalima automatski unosili kontinuirani očekivani trošak uz pravovremeni podsjetnik o trošku, bilo da se radi o plaćanju kredita, pristojbi, članarina ili kupovini na rate.
- Mogućnost odabira postavljenog jezika između hrvatskog i engleskog.
- Unos troškova u različitim valutama (euro i dolar), čiji bi se iznosi zatim preračunavali u korisnikovu početno zadanu valutu.
- Fotografiranje računa i učitavanje u aplikaciju. Na taj način svi podaci o rashodima nalazili bi se na jednom, zaštićenom mjestu.
- Sinkronizacija financijskih podataka s bankarskim računom korisnika. U periodu razvoja aplikacije, takvu značajku nije bilo moguće ostvariti jer je aplikacija u vlasništvu privatne osobe. Iako je Direktiva o platnim uslugama (PSD2) stupila na snagu početkom 2018. godine, njezina konkretna primjena započela je krajem 2019 godine [36]. Sukladno tome, bankarski sustavi počeli su pružati jedinstven pristup otvorenom API sučelju drugim autoriziranim organizacijama i tvrtkama tj. licenciranim pružateljima platnih usluga [37].

Zaključak

Iako na tržištu postoje brojne slične aplikacije za vođenje osobnih financija, malo ih je napravljeno u najnovijim tehnologijama i ne iskorištavaju njihov puni potencijal. Većina ih ima zastarjelo korisničko sučelje te zbog ograničenja tehnologije u kojoj su razvijene ne mogu pružiti korisnicima funkcionalnosti koje im trebaju. Mogućnosti za nadogradnje i proširenja starih sustava su ograničene jer bi to zahtijevalo potpuno prepisivanje u nove tehnologije što često nije moguće.

Krajnji rezultat ovog rada je cjeloviti raspodijeljeni sustav za kontrolu i vođenje osobnih financija napravljen u najnovijim tehnologijama. Sastoji se od jednog poslužitelja koji predstavlja Web API, a izgrađen je u ASP.NET Core 3 tehnologiji te dva klijenta – web i mobilne aplikacije koje se spajaju na API i prikazuju podatke. Web aplikacija izrađena je u programskom okviru Vue.js, a mobilna u NativeScript-Vue-u koji podržava rad aplikacije na svim mobilnim platformama. Namijenjen je svima koji žele voditi brigu o svojim osobnim financijama na jednostavan i pristupačan način u digitalnom obliku.

Unatoč postojanju brojnih sličnih sustavu na tržištu, razvijeni sustav bi uz svoje različitosti, ali i prednosti, mogao pronaći svoje mjesto kod korisnika. Nudi mogućnost jednostavnog upravljanja osobnim financijama, neograničenu geolokacijsku upotrebljivost, mnoštvo grafova za poboljšanje korisničkog iskustva te dostupnost na gotovo svim platformama i uređajima. Korisničko sučelje je dizajnirano tako da bude intuitivno uz maksimalno olakšano korištenje aplikacije. Uz dodatne nadogradnje poput spremanja slika računa i podrškom za više valuta, mogao bi postati dobar konkurent već postojećim rješenjima na regionalnom tržištu.

Popis kratica

AJAX	*Asynchronous JavaScript + XML*	asinkroni JavaScript i XML označni jezik
API	*Application programming interface*	aplikacijsko programsko sučelje
CLI	*Command line interface*	sučelje komandne linije
CSS	*Cascading Style Sheets*	kaskadni stilski listovi
DAL	*Data Access Layer*	sloj pristupa podacima
DIP	*Dependency inversion principle*	princip inverzije ovisnosti
DOM	*Document object model*	model objekta dokumenta
DTO	*Data Transfer Object*	objekt za prijenos podataka
HTML	*HyperText Markup Language*	tekstualni označni jezik
HTTP	*HyperText transfer protocol*	protokol prijenos informacija
HTTPS	*HyperText Transfer Protocol Secure*	osigurani protokol prijenosa informacija
IDE	*Interactive development environment*	interaktivno razvojno okruženje
MVC	*Model-view-controller*	model-pogled-upravitelj
NPM	*Node package manager*	upravitelj paketima Node.js biblioteka
ORM	*Object Relational Mapping*	objektno relacijsko mapiranje
PMC	*Package Manager Console*	konzola za upravljanje paketima
REST	*Representational state transfer*	reprezentativni prijenos stanja
SDLC	*Software development life cycle*	životni ciklus razvoja softvera
SPA	*Single Page Application*	jednostrana aplikacija
SQL	*Structured Query Language*	strukturni upitni jezik
URL	*Uniform Resource Locator*	jedinstveni lokator resursa
YNAB	*You Need A Budget*	

Popis slika

Popis tablica

Popis kôdova

Literatura

[1] CVRLJE, D. PRIRUČNIK ZA VJEŽBE IZ KOLEGIJA OSOBNE FINANCIJE. ZAGREB: EKONOMSKI FAKULTET, 2015.

[2] MUNOSHAMY, T. PERSONAL FINANCIAL MANAGEMENT. 2012.

[3] https://www.statista.com/statistics/954622/types-of-financial-apps-used-by-americans, OPIS APLIKACIJA ZA UPRAVLJANJE NOVCEM, PROSINAC. 2019.

[4] https://www.doughroller.net/budgeting/10-tablet-apps-that-manage-your-money/, OPIS APLIKACIJA ZA UPRAVLJANJE NOVCEM, PROSINAC. 2019.

[5] https://www.thebalance.com/top-10-budget-software-apps-1293609, OPIS APLIKACIJA ZA UPRAVLJANJE NOVCEM, PROSINAC. 2019.

[6] https://www.thebalance.com/best-expense-tracker-apps-4158958, OPIS APLIKACIJA ZA UPRAVLJANJE NOVCEM, PROSINAC. 2019.

[7] www.magnifymoney.com/blog/banking-apps/review-toshl-finance-budgeting-app1051582924/, OPIS TOSHL APLIKACIJE, PROSINAC. 2019.

[8] LJUBI, I. MIHALJEVIĆ, B. INTEROPERABILNOST INFORMACIJSKIH SUSTAVA: PRIRUČNIK. ZAGREB: ALGEBRA, 2013.

[9] KAŠTELAN, T. UVOD U BAZE PODATAKA: PRIRUČNIK. ZAGREB: ALGEBRA, 2010.

[10] http://files.fpz.hr/Djelatnici/tcaric/Tonci-Caric-Baze-podataka.pdf, DEFINICIJA BAZE PODATAKA, STR. 2, PROSINAC. 2019.

[11] https://www.algebra.hr/cjelozivotno-obrazovanje/wp-content/uploads/sites/3/2017/12/Administrator-baza-podataka.pdf, TABLICE U BAZI PODATAKA, STR. 6, PROSINAC. 2019.

[12] https://www.learnentityframeworkcore.com, DEFINICIJA ENTITY FRAMEWORK-A, SIJEČANJ. 2020.

[13] https://docs.microsoft.com/en-us/ef/core/ DEFINICIJA ENTITY FRAMEWORK-A, SIJEČANJ. 2020.

[14] https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli, EF CORE MIGRACIJE, SIJEČANJ. 2020.

[15] http://www.binaryintellect.net/articles/5e180dfa-4438-45d8-ac78-c7cc11735791.aspx, INICIJALNO POSTAVLJANJE PODATAKA U MICROSOFT IDENTITY-U, SIJEČANJ. 2020.

[16] https://www.learnentityframeworkcore.com/dbset, DEFINICIJA DBSET-A, SIJEČANJ. 2020.

[17] MILLET, S. PROFESSIONAL ASP.NET DESIGN PATTERNS. INDIANAPOLIS: WILEY PUBLISHING, INC., 2010.

[18] https://garywoodfine.com/generic-repository-pattern-net-core/, OBRAZAC REPOZITORIJA I JEDINICE-RADA U ENTITY FRAMEWORK-U, SIJEČANJ. 2020.

[19] https://martinfowler.com/eaaCatalog/serviceLayer.html, SERVISNI SLOJ, SIJEČANJ. 2020.

[20] https://code-maze.com/global-error-handling-aspnetcore/, GLOBALNO HVATANJE POGREŠAKA, SIJEČANJ. 2020.
[21] https://www.codecademy.com/articles/what-is-rest, DEFINICIJA REST-A, SIJEČANJ. 2020.
[22] https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f, DEFINICIJA REST-A, SIJEČANJ. 2020.
[23] https://medium.com/codingthesmartway-com-blog/getting-started-with-axios-166cb0035237, SVRHA AXIOS BIBLIOTEKE, siječanj. 2020.
[24] https://code-maze.com/vuejs-axios-http-environment-files/, KONFIGURACIJA AXIOS BIBLIOTEKE, SIJEČANJ. 2020.
[25] https://v1.vuejs.org/guide/components.html, VUE.JS KOMPONENTE, SIJEČANJ 2020.
[26] https://medium.com/learning-lab/lessons-learned-on-writing-apps-with-nativescript-vuejs-bd6a3066f0cb, RAZVOJ MOBILNIH APLIKACIJA NATIVESCRIPT-OM, SIJEČANJ. 2020.
[27] https://nativescript-vue.org/en/docs/introduction/, DEFINICIJA NATIVESCRIPT-VUE-A, SIJEČANJ. 2020.
[28] https://medium.com/learning-lab/lessons-learned-on-writing-apps-with-nativescript-vuejs-bd6a3066f0cb, JEDNOSTAVNO KOMANDNO SUČELJE (CLI), SIJEČANJ. 2020.
[29] https://medium.com/@jamie_45704/nativescript-vue-creating-a-global-side-drawer-fd1c76683722, OPIS SIDE DRAWER KOMPONENTE, SIJEČANJ. 2020.
[30] https://docs.nativescript.org/vuejs/ns-ui/Chart/overview#chart-overview, RADPIECHART, SIJEČANJ. 2020.
[31] https://docs.nativescript.org/vuejs/ns-ui/Chart/Series/Types/donut, DONUTSERIES, SIJEČANJ. 2020.
[32] https://docs.microsoft.com/en-us/aspnet/core/security/?view=aspnetcore-3.1, DEFINICIJA AUTORIZACIJE I AUTENTIFIKACIJE, SIJEČANJ. 2020.
[33] https://www.freecodecamp.org/news/restful-services-part-i-http-in-a-nutshell-aab3bfedd131/, HTTPS MREŽNI PROTOKOL, SIJEČANJ. 2020.
[34] https://docs.microsoft.com/en-us/dotnet/architecture/microservices/secure-net-microservices-web-applications/, SIGURNOST WEB APLIKACIJA, SIJEČANJ. 2020.
[35] https://garywoodfine.com/asp-net-core-2-2-jwt-authentication-tutorial/, JSON web token, DEFINICIJA JSON WEB TOKENA, SIJEČANJ. 2020.
[36] https://www.hub.hr/hr/sto-je-psd2-direktiva, PRIMJENA PSD2 DIREKTIVE, SIJEČANJ. 2020.
[37] https://www.erstebank.hr/hr/press/priopcenja-za-medije/2019/6/11/spremni-za-psd2-i-otvoreno-bankarstvo-erste-group-pruza-jedinstven-pristup-api-suceljima-svih-svojih-podruznica-u-istocnom-dijelu-eu-a, PRIMJENA PSD2 DIREKTIVE, SIJEČANJ. 2020.