Alat za predviđanje i upravljanje osobnim financijama

Buljan, Filip

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1287

ALAT ZA PREDVIĐANJE I UPRAVLJANJE OSOBNIM FINANCIJAMA

Filip Buljan

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1287

ALAT ZA PREDVIĐANJE I UPRAVLJANJE OSOBNIM FINANCIJAMA

Filip Buljan

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1287

Pristupnik:	**Filip Buljan (0036539840)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Ivica Botički

Zadatak: **Alat za predviđanje i upravljanje osobnim financijama**

Opis zadatka:

U okviru završnog rada potrebno je razviti alat za upravljanje osobnim financijama i predviđanje budućih financijskih stanja. Alat treba korisnicima omogućiti bolje razumijevanje i upravljanje vlastitim financijama te pomoći u ostvarivanju dugoročnih financijskih ciljeva kroz bazične modele predviđanja. Korisnici putem alata na siguran i jednostavan način unose svoje financijske podatke, postavljaju osobne financijske ciljeve (npr. štednja za odmor, otplata duga) i primaju personalizirane izvještaje i preporuke za optimizaciju financija. Sustav uključuje relacijsku bazu podataka i poslužiteljsku komponentu u okviru koje će se implementirati razni oblikovni obrasci za osiguranje skalabilnosti i održivosti koda. Klijentski dio sustava uključuje razvoj korisničkog sučelja za unos podataka i prikaz izvještaja i predviđanja.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

Uvod

U današnjem užurbanom i financijski zahtjevnom svijetu, praćenje osobnih financija postalo je ključno za osiguranje financijske stabilnosti i dugoročnog planiranja. Mnogi pojedinci suočavaju se s izazovima u upravljanju svojim prihodima, troškovima, štednjom i investicijama, što često dovodi do financijskog stresa i neizvjesnosti. Iz tog razloga, nastala je potreba za aplikacijom koja će omogućiti jednostavno i efikasno upravljanje osobnim financijama.

Aplikacija PennyPlanner razvijena je kako bi zadovoljila ovu potrebu pružajući korisnicima alat za organizaciju, planiranje i praćenje osobnih financija. Cilj aplikacije je pomoći korisnicima da bolje razumiju i kontroliraju svoje financijske tokove, postavljaju financijske ciljeve te donose informirane odluke o svojoj potrošnji i štednji.

PennyPlanner omogućuje korisnicima kreiranje i upravljanje različitim računima, bilježenje i kategorizaciju transakcija te postavljanje financijskih ciljeva. Frontend aplikacije izrađen je korištenjem Vue3 i Vuetify3 uz JavaScript, dok je backend razvijen kao ASP.NET Core Web API aplikacija koristeći C#. Podaci se pohranjuju u relacijskoj bazi podataka SQLite.

Ovaj rad pružit će detaljan uvid u motive za razvoj aplikacije PennyPlanner, korištene tehnologije, arhitekturu sustava te funkcionalnosti koje aplikacija nudi korisnicima. Kroz analizu i opis implementacije, cilj je demonstrirati kako ova aplikacija može doprinijeti boljem upravljanju osobnim financijama i poboljšanju financijske pismenosti korisnika.

1. Korištene tehnologije

1.1. ASP.NET Core

ASP.NET Core je moderni, visoko performantni framework za izradu web aplikacija i API-ja, razvijen od strane Microsofta [1]. Ovaj framework omogućuje razvoj aplikacija koje se mogu pokretati na različitim platformama, kao što su Windows, macOS i Linux, te podržava modularnu i fleksibilnu arhitekturu. ASP.NET Core je otvorenog koda i dio .NET ekosustava, što ga čini idealnim za razvoj raznovrsnih web rješenja.

Web API (Application Programming Interface) omogućuje komunikaciju između različitih softverskih komponenti putem HTTP protokola. ASP.NET Core Web API omogućuje izradu RESTful API-ja [2], koji klijentima omogućuju pristup funkcionalnostima aplikacije putem HTTP zahtjeva [3].

Osnovne značajke ASP.NET Core Web API-ja uključuju modularnost i ekstenzibilnost, korištenje posredničkog sloja komponenti za obradu HTTP zahtjeva i odgovora, ugrađenu podršku za dependency injection, fleksibilnu konfiguraciju aplikacije, te robusne mehanizme za autentifikaciju i autorizaciju korisnika. Ove značajke omogućuju izradu sigurnih, skalabilnih i lako održivih aplikacija.

ASP.NET Core koristi modularni dizajn koji omogućuje dodavanje samo onih komponenti koje su potrebne za aplikaciju. Ovo smanjuje veličinu aplikacije i poboljšava performanse. Posrednički sloj je komponenta koja se koristi za obradu HTTP zahtjeva i odgovora. U ASP.NET Core-u, komponente posredničkog sloja se konfiguriraju u Program.cs datoteci, omogućujući prilagodbu cijelog tijeka obrade zahtjeva.

ASP.NET Core ima ugrađenu podršku za dependency injection, što olakšava upravljanje ovisnostima između komponenti. Ovaj pristup potiče razvoj aplikacija koje su lakše za testiranje i održavanje. Konfiguracija aplikacije u ASP.NET Core-u je fleksibilna i podržava različite izvore, kao što su JSON datoteke, varijable okruženja i tajne (eng. *secrets*). To omogućuje lako upravljanje postavkama aplikacije u različitim okruženjima.

ASP.NET Core pruža robusne mehanizme za autentifikaciju i autorizaciju korisnika. Može se koristiti različite metode autentifikacije, uključujući JWT (JSON Web Token)

autentifikaciju, koja je prikladna za moderne web aplikacije i API-je. Logging je integriran u ASP.NET Core i omogućuje jednostavno praćenje i bilježenje aktivnosti unutar aplikacije.

U PennyPlanner aplikaciji, ASP.NET Core Web API koristi se za backend komponentu koja upravlja podacima i poslovnom logikom aplikacije. Ključne komponente backend arhitekture uključuju modele, kontrolere, servise i repozitorije. Aplikacija koristi relacijsku bazu podataka SQLite za pohranu podataka, a za logiranje se koristi Serilog.

U Program.cs datoteci konfiguriran je cijeli tijek obrade zahtjeva, uključujući registraciju potrebnih servisa, postavljanje baze podataka, konfiguraciju posredničkog sloja za logiranje i rukovanje iznimkama, te postavljanje autentifikacije i autorizacije koristeći JWT. Ova konfiguracija omogućuje sigurnu i efikasnu obradu zahtjeva te osigurava da aplikacija radi kako je predviđeno u različitim okruženjima.

1.2. SQLite

SQLite je popularna, relacijska baza podataka koja se koristi u brojnim aplikacijama zbog svoje jednostavnosti i učinkovitosti. Za razliku od tradicionalnih SQL baza podataka koje rade kao samostalni poslužitelji, SQLite je integrirana baza podataka koja se sprema u datoteku na lokalnom sustavu. Ova karakteristika čini je izuzetno pogodnom za aplikacije koje trebaju laganu bazu podataka bez složene konfiguracije i održavanja [4].

SQLite se često koristi u mobilnim aplikacijama, embedded sustavima, te manjim desktop i web aplikacijama, gdje jednostavnost i brzina pristupa podacima imaju prioritet. Unatoč svojoj jednostavnosti, SQLite podržava mnoge napredne SQL značajke, uključujući transakcije, podupiruće ključeve i složene upite.

Osnovne značajke SQLite uključuju visoku učinkovitost, jednostavnu integraciju i nisku potrebu za održavanjem. SQLite baze podataka su jednostavne za postavljanje i ne zahtijevaju instalaciju ili konfiguraciju poslužitelja. Sve što je potrebno je datoteka baze podataka koja se može koristiti direktno unutar aplikacije. Ova pristupačnost čini SQLite idealnim izborom za aplikacije koje trebaju samo pohranu podataka bez dodatnih složenosti.

U aplikaciji PennyPlanner, SQLite se koristi kao relacijska baza podataka za pohranu i upravljanje podacima korisnika, računa, transakcija i ciljeva. Korištenje SQLite-a omogućava brzo i efikasno upravljanje podacima, dok istovremeno pojednostavljuje proces postavljanja i održavanja baze podataka.

1.3. Entity Framework

Entity Framework Core (EF Core) je Object-Relational Mapping (ORM) framework razvijen od strane Microsofta, koji omogućava rad s bazama podataka koristeći .NET objekte. EF Core omogućuje razvoj aplikacija koristeći objektno-orijentirani pristup, apstrahirajući složenost rada s relacijskim bazama podataka. Ovaj pristup olakšava upravljanje podacima i omogućuje brži razvoj aplikacija [5].

Osnovne značajke EF Core uključuju podršku za različite baze podataka, kao što su SQL Server, SQLite, PostgreSQL i druge, mogućnost mapiranja .NET objekata na tablice u bazi podataka, te podršku za složene upite i transakcije. EF Core također omogućuje migracije baza podataka, što olakšava upravljanje promjenama u strukturi baze podataka tijekom razvoja aplikacije.

U aplikaciji PennyPlanner, EF Core se koristi za mapiranje modela aplikacije na tablice u SQLite bazi podataka. Korištenje EF Core-a omogućuje razvijanje aplikacije koristeći poznatu C# biblioteku System.Linq, dok se SQL upiti i upravljanje bazom podataka apstrahiraju. Ovo olakšava razvoj, održavanje i proširenje aplikacije.

Kontekst baze podataka definiran je u klasi ApplicationDbContext, koja nasljeđuje IdentityDbContext<IdentityUser, IdentityRole, string>. Ova klasa sadrži DBSet<Entitet> kolekcije koje predstavljaju tablice u bazi podataka, kao što su korisnici, računi, ciljevi i transakcije. Primjer implementacije konteksta baze podataka prikazan je slikom (Slika 1.1).

```
public class ApplicationDbContext : IdentityDbContext<IdentityUser, IdentityRole, string>
{
    0 references
    public DbSet<User> Users { get; set; }
    0 references
    public DbSet<Account> Accounts { get; set; }
    0 references
    public DbSet<Goal> Goals { get; set; }
    0 references
    public DbSet<Transaction> Transactions { get; set; }

    0 references
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    {

    }

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Filename=Database.db");
        base.OnConfiguring(optionsBuilder);
    }

    0 references
    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.Entity<User>().HasKey(e => e.Id);
        builder.Entity<Account>().HasKey(e => e.Id);
        builder.Entity<Goal>().HasKey(e => e.Id);
        builder.Entity<Transaction>().HasKey(e => e.Id);

        builder.Entity<User>()
            .HasMany(e => e.Accounts)
            .WithOne(e => e.User);

        builder.Entity<User>()
            .HasMany(e => e.Goals)
            .WithOne(e => e.User);

        builder.Entity<User>()
            .HasMany(e => e.Transactions)
            .WithOne(e => e.User);

        builder.Entity<Account>()
            .HasMany(e => e.Transactions)
            .WithOne(e => e.Account);

        base.OnModelCreating(builder);
    }
}
```

Slika 1.1 Implementacija konteksta baze podataka

1.4. Vue 3

Vue 3 je napredni JavaScript framework za izradu korisničkih sučelja, razvijen od strane Evana You i open-source zajednice. Poznat po svojoj fleksibilnosti, jednostavnosti korištenja i visokim performansama, Vue 3 omogućuje razvoj dinamičnih i responzivnih web aplikacija. Ovaj framework koristi reaktivni model podataka, komponentnu arhitekturu i virtualni DOM za učinkovito ažuriranje sučelja [6].

Osnovne značajke Vue 3 uključuju komponente, reaktivnost, templating sustav, direktive i napredne tehnike optimizacije. Komponente omogućuju razbijanje korisničkog sučelja na manje, ponovno upotrebljive dijelove. Reaktivnost osigurava da promjene u podacima automatski osvježavaju korisničko sučelje. Templating sustav omogućuje deklarativno definiranje strukture sučelja, dok direktive omogućuju manipulaciju DOM-om na deklarativan način.

U aplikaciji PennyPlanner, Vue 3 se koristi za izradu frontenda. Korištenje Vue 3 omogućuje razvoj intuitivnog i responzivnog korisničkog sučelja koje korisnicima omogućuje jednostavno upravljanje osobnim financijama. Vue 3 također olakšava razvoj modularnih i održivih komponenti koje se mogu lako proširivati i prilagođavati.

Integracija Vue 3 u PennyPlanner aplikaciji započinje s postavljanjem osnovne strukture aplikacije koristeći Vue CLI (Command Line Interface). Vue CLI pruža alate za brzo generiranje projekta i postavljanje potrebnih ovisnosti. Nakon inicijalne postavke, aplikacija se može strukturirati koristeći komponente, svaka odgovorna za određeni dio korisničkog sučelja.

U Vue 3, svaka komponenta se obično sastoji od tri glavna dijela: HTML, CSS i JavaScript. HTML dio komponente definira strukturu korisničkog sučelja i koristi Vue-ov templating sustav za povezivanje podataka s prikazom. CSS dio služi za stiliziranje komponente, dok JavaScript dio sadrži logiku i podatke komponente. Ovi dijelovi su organizirani unutar <template>, <style> i <script> tagova, što olakšava održavanje i razumijevanje koda.

Vue 3 podržava dva različita pristupa za definiranje komponenata: Options API i Composition API. Options API je tradicionalni pristup koji se koristi u starijim verzijama Vue-a i organizira kod unutar objekta s ključevima kao što su data, methods, computed i watch. Ovaj pristup je jednostavan za razumijevanje i koristi se u aplikaciji PennyPlanner.

Composition API je noviji pristup uveden u Vue 3, koji omogućuje fleksibilniji i modularniji način definiranja komponenata. Umjesto organiziranja koda unutar jednog objekta, Composition API koristi funkcije i reaktivne reference kako bi se definirale logika i podaci komponente.

U aplikaciji PennyPlanner koristi se Options API, koji je tradicionalan i široko prihvaćen pristup za definiranje Vue komponenti. Primjer implementacije JavaScript dijela jedne Vue komponente kada se koristi Options API može se vidjeti na slici (Slika 1.2).

Slika 1.2 Primjer implementacije skripte koristeći Options API

1.4.1. Vuetify 3

Vuetify 3 je popularna Vue.js biblioteka komponenti koja implementira Material Design specifikacije razvijene od strane Googlea. Pruža bogat skup UI komponenti koje su dizajnirane da budu estetski privlačne, responzivne i jednostavne za korištenje. Vuetify

omogućuje brzi razvoj modernih web aplikacija s dosljednim i profesionalnim korisničkim sučeljem [7].

Osnovne značajke Vuetify 3 uključuju unaprijed definirane komponente poput obrazaca, tipografije, navigacijskih elemenata, dijaloga, tablica i mnogih drugih. Svaka komponenta je prilagodljiva i može se jednostavno integrirati u Vue.js aplikaciju. Vuetify također podržava teme, omogućujući razvoj aplikacija s prilagođenim stilovima koji odgovaraju specifičnim brendovima ili dizajnerskim smjernicama.

Integracija Vuetify 3 u aplikaciji PennyPlanner omogućuje brzo stvaranje modernog i responzivnog korisničkog sučelja. Korištenje Vuetify komponenata pojednostavljuje razvoj kompleksnih UI elemenata i osigurava dosljedan izgled cijele aplikacije. Vuetify komponente su prilagodljive i mogu se lako proširiti prema specifičnim potrebama aplikacije.

2. Arhitektura sustava

Arhitektura sustava aplikacije PennyPlanner se sastoji od tri glavna sloja: poslužiteljski sloj (eng. *backend*), klijentski sloj (eng. *frontend*) i baza podataka. Svaki od ovih slojeva ima specifične odgovornosti i koristi različite tehnologije kako bi osigurao optimalno funkcioniranje aplikacije.

To zapravo predstavlja MVC (Model-View-Controller) način arhitekture, gdje su odgovornosti podijeljene između modela, pogleda i kontrolera. Modeli upravljaju podacima i poslovnom logikom, pogledi su zaduženi za prikaz korisničkog sučelja, dok kontroleri upravljaju protokom podataka između modela i pogleda te obrađuju korisničke zahtjeve [8].

Poslužiteljski sloj implementiran je koristeći ASP.NET Core Web API dok je klijentski sloj razvijen koristeći Vue 3 i Vuetify 3, a baza podataka temelji se na SQLite.

2.1. Poslužiteljski sloj

Poslužiteljski sloj aplikacije PennyPlanner oblikovan je prema obrascu Controller-Service-Repository, koji osigurava jasnu podjelu odgovornosti i olakšava održavanje koda. Kontroleri su zaduženi za primanje i obradu API zahtjeva od klijenta. Oni referenciraju servise koji implementiraju poslovnu logiku aplikacije, a servisi zatim komuniciraju s repozitorijima koji služe za pristup i upravljanje podacima u bazi podataka.

Poslužiteljski sloj također koristi komponente posredničkog sloja koje dodatno poboljšavaju funkcionalnost i sigurnost aplikacije. Jedna od ključnih komponenti posredničkog sloja je komponenta za obradu iznimki, koja osigurava da se sve pogreške pravilno evidentiraju i obrade. Druga komponenta posredničkog sloja služi za validaciju podataka koristeći paket FluentValidation te osigurava da svi zahtjevi sadrže ispravne i potpune informacije prije nego što se obrade. Također je implementiran i jednostavan ugrađeni dio posredničkog sloja za zapisivanje u konzolu i log datoteke, što pomaže u praćenju rada aplikacije i dijagnostici problema.

Za mapiranje podataka između različitih slojeva aplikacije koristi se AutoMapper. Ovaj alat omogućuje automatsko mapiranje objekata, čime se smanjuje potreba za ručnim

kodiranjem mapiranja i smanjuje mogućnost pogrešaka. AutoMapper se koristi za pretvaranje entiteta baze podataka u DTO-ove (Data Transfer Objects) koji se koriste u komunikaciji između poslužiteljskog sloja i klijentskog sloja.

Da bi se osigurala pravilna upravljivost svih ovih komponenti i njihova integracija unutar aplikacije, koristi se obrazac Dependency Injection (DI).

DI je oblikovni obrazac koji omogućuje lakše upravljanje ovisnostima unutar aplikacije [9]. U ASP.NET Core, DI je ključna značajka koja omogućuje stvaranje i upravljanje životnim ciklusom objekata te njihovo umetanje u klase koje ih koriste. Ovaj pristup poboljšava modularnost, testabilnost i održavanje aplikacije.

U aplikaciji PennyPlanner, DI se koristi za registraciju i upravljanje servisima, validatorima i drugim komponentama. Registracija servisa se obavlja u klasi DIConfiguration.cs, gdje se definiraju sve potrebne ovisnosti. Ova konfiguracija se zatim koristi tijekom pokretanja aplikacije kako bi se osiguralo da su svi potrebni servisi dostupni unutar cijele aplikacije.

```
public class DIConfiguration
{
    1 reference
    public static void RegisterServices(IServiceCollection services)
    {
        services.AddAutoMapper(typeof(DtoEntityMapperProfile));
        services.AddScoped<IUserService, UserService>();
        services.AddScoped<IAuthService, AuthService>();
        services.AddScoped<IAccountService, AccountService>();
        services.AddScoped<ITransactionService, TransactionService>();

        services.AddScoped<UserCreateValidator>();
        services.AddScoped<UserUpdateValidator>();
        services.AddScoped<TransactionCreateValidator>();
        services.AddScoped<TransactionUpdateValidator>();
    }
}
```

Slika 2.1 Implementacija DIConfiguration.cs

U primjeru na slici (Slika 2.1), DIConfiguration klasa registrira različite servise kao što su UserService, AuthService, AccountService i TransactionService. Također su registrirani validatori za stvaranje i ažuriranje korisnika i transakcija. Registracija servisa koristi AddScoped metodu koja osigurava da se novi primjerak servisa stvara za svaki zahtjev.

2.1.1. Kontroleri

Kontroleri u aplikaciji PennyPlanner igraju ključnu ulogu u poslužiteljskom sloju, služeći kao posrednici između korisničkih zahtjeva i poslovne logike implementirane u servisima.

Svaki kontroler je odgovoran za upravljanje specifičnim entitetima unutar aplikacije, kao što su računi, transakcije, korisnici i ciljevi.

Kontroleri koriste atribut [ApiController] za označavanje klase kao kontrolera API-ja, te [Route] atribut za definiranje ruta koje odgovaraju HTTP zahtjevima. Ovo omogućuje jasnu organizaciju i pristup različitim funkcionalnostima aplikacije putem dobro definiranih API krajnjih točaka.

Primarna odgovornost kontrolera je primanje dolaznih HTTP zahtjeva te prosljeđivanje podataka odgovarajućim servisima koji implementiraju poslovnu logiku. Nakon što servisi obrade zahtjeve, kontroleri vraćaju odgovarajuće HTTP odgovore klijentu. Kontroleri također osiguravaju da su svi odgovori standardizirani i konzistentni.

U aplikaciji PennyPlanner, kontroleri su implementirani za sve entitete: računi (eng. account), transakcije (eng. *transaction*), korisnici (eng. *user*) i ciljevi (eng. *goal*). Na primjer, AccountController upravlja operacijama vezanim uz račune, uključujući kreiranje, ažuriranje, brisanje i dohvaćanje računa. Svaka metoda unutar kontrolera odgovara specifičnoj operaciji te jednoj API krajnjoj točci i koristi odgovarajuće HTTP metode kao što su POST, PUT, DELETE i GET. Primjer njegove implementacije vidljiv je na slici (Slika 2.2).

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class AccountController : ControllerBase
{
    8 references
    private IAccountService AccountService { get; }

    0 references
    public AccountController(IAccountService accountService)
    {
        AccountService = accountService;
    }

    [Authorize]
    [HttpPost("create")]
    0 references
    public async Task<IActionResult> CreateAccount(AccountCreate accountCreate)
    {
        var id = await AccountService.CreateAccountAsync(accountCreate);
        var account = await AccountService.GetAccountAsync(id);
        var response = new { success = true, account };

        return Ok(response);
    }

    [Authorize]
    [HttpPut("update")]
    0 references
    public async Task<IActionResult> UpdateAccount(AccountUpdate accountUpdate)
    {
        await AccountService.UpdateAccountAsync(accountUpdate);
        var account = await AccountService.GetAccountAsync(accountUpdate.Id);

        return Ok(account);
    }

    [Authorize]
    [HttpDelete("delete")]
    0 references
    public async Task<IActionResult> DeleteAccount(AccountDelete accountDelete)
    {
        await AccountService.DeleteAccountAsync(accountDelete);
        return Ok();
    }

    [Authorize]
    [HttpGet("get/{id}")]
    0 references
    public async Task<IActionResult> GetAccount(int id)
    {
        var account = await AccountService.GetAccountAsync(id);
        return Ok(account);
    }
```

Slika 2.2 Implementacija UserController.cs

Kontroleri također koriste dependency injection za pristup servisima. Ovo osigurava da su servisi pravilno instancirani i da se njihove ovisnosti mogu lako upravljati.

Osim osnovne funkcionalnosti, kontroleri mogu koristiti atribut [Authorize] za zaštitu osjetljivih podataka i operacija. Ovo osigurava da samo ovlašteni korisnici mogu pristupiti određenim funkcionalnostima aplikacije, što povećava sigurnost sustava, a to radi na način da provjerava prisutnost autorizacijskog zaglavlja u nadolazećem API zahtjevu te ispravnost JWT tokena u sadržaju tog zaglavlja.

2.1.2. Servisi

Servisi u aplikaciji PennyPlanner čine sloj koji implementira poslovnu logiku i služi kao posrednik između kontrolera i repozitorija. Svaki servis se sastoji od sučelja (eng. *interface*) i implementacijske klase. Sučelje definira ugovor (eng. *contract*) koji specificira koje metode servis mora implementirati, dok implementacijska klasa sadrži stvarnu logiku koja izvršava te metode.

Korištenje sučelja i implementacijskih klasa pruža nekoliko prednosti. Prvo, omogućuje jasnu separaciju odgovornosti, gdje sučelje definira što servis treba raditi, a implementacijska klasa definira kako to radi. Ovo olakšava testiranje i zamjenu implementacija, jer se testovi mogu pisati protiv sučelja, a ne konkretnih implementacija. Drugo, ovo omogućuje lako proširivanje i održavanje koda, jer promjene u implementaciji ne utječu na sučelje, a samim time ni na dijelove aplikacije koji koriste servis preko sučelja.

Svaki servis u PennyPlanner aplikaciji je registriran u sustavu za dependency injection, što omogućuje jednostavno upravljanje životnim ciklusom servisa i njihovih ovisnosti. Servisi se koriste za razne operacije kao što su kreiranje, ažuriranje, brisanje i dohvaćanje podataka iz baze, uz primjenu poslovne logike specifične za aplikaciju.

Na primjer, UserService upravlja operacijama vezanim uz korisnike. Sučelje IUserService definira metode potrebne za kontrolu korisnika te podataka usko vezanih uz njih. Sučelje je prikazano na slici (Slika 2.3). Implementacijska klasa UserService implementira ove metode koristeći repozitorij za pristup podacima, validatore za provjeru ulaznih podataka i AutoMapper za mapiranje podataka između različitih modela.

Ovaj pristup omogućuje da poslovna logika bude jasno definirana i izolirana u servisima, dok kontroleri ostaju fokusirani na upravljanje HTTP zahtjevima i vraćanje odgovora klijentima. Servisi također omogućuju ponovno korištenje poslovne logike na različitim mjestima unutar aplikacije, čime se smanjuje dupliciranje koda i povećava održivost.

```
public interface IUserService
{
    2 references
    Task<int> CreateUserAsync(UserCreate userCreate);
    2 references
    Task UpdateUserAsync(UserUpdate userUpdate);
    2 references
    Task DeleteUserAsync(UserDelete userDelete);
    4 references
    Task<UserGet> GetUserAsync(int id);
    3 references
    Task<List<UserGet>> GetUsersAsync();
    2 references
    Task<UserGet?> GetUserByLoginAsync(string login);
}
```

Slika 2.3 Sučelje IUserService.cs

2.1.3. Repozitoriji

Repozitoriji služe za pristup i upravljanje podacima u bazi podataka. Oni predstavljaju sloj između servisa i baze podataka, omogućujući izolaciju poslovne logike od konkretnih implementacija za pristup podacima. Repozitoriji pružaju apstrakciju koja omogućuje lako mijenjanje implementacije pristupa podacima bez utjecaja na ostatak aplikacije.

U aplikaciji PennyPlanner, repozitoriji su implementirani s pomoću generičkog repozitorija. Generički repozitorij je ponovno iskoristiv za bilo koji entitet u aplikaciji, čime se smanjuje dupliciranje koda i pojednostavljuje održavanje. Generički repozitorij omogućuje zajedničke operacije kao što su dohvaćanje, umetanje, ažuriranje i brisanje podataka za sve entitete. Takav repozitorij je definiran s pomoću sučelja IGenericRepository<T>, gdje T predstavlja tip entiteta koji repozitorij upravlja. Ovo sučelje definira metode za dohvaćanje podataka s filtrima, dohvaćanje podataka po identifikatoru, umetanje, ažuriranje, brisanje i spremanje promjena. Izgled tog sučelja vidljiv je na slici (Slika 2.4).

```
public interface IGenericRepository<T> where T : BaseEntity
{
    1 reference
    Task<List<T>> GetFilteredAsync(Expression<Func<T, bool>>[] filters, int? skip, int? take, params Expression<Func<T, object>>[] includes);
    6 references
    Task<List<T>> GetAsync(int? skip, int? take, params Expression<Func<T, object>>[] includes);
    13 references
    Task<T?> GetByIdAsync(int id, params Expression<Func<T, object>>[] includes);
    4 references
    Task<int> InsertAsync(T entity);
    4 references
    void Update(T entity);
    4 references
    void Delete(T entity);
    11 references
    Task SaveChangesAsync();
}
```

Slika 2.4 Sučelje IGenericRepository.cs

2.1.4. Posrednički sloj

Posrednički sloj (eng. *middleware*) je skup komponenata u ASP.NET Core koji se koristi za obradu HTTP zahtjeva i odgovora u tijeku njihove obrade kroz aplikaciju. Komponente se postavljaju u cjevovod obrade zahtjeva i mogu izvršavati različite zadatke, kao što su autentifikacija, logiranje, validacija i rukovanje iznimkama. U aplikaciji PennyPlanner koristi se nekoliko ključnih middleware komponenti kako bi se osigurala sigurnost, validacija i praćenje rada aplikacije. U nastavku se nalazi objašnjenje svake middleware komponente koja je eksplicitno implementirana.

Prva komponenta koja se koristi je komponenta za obradu iznimaka (eng. *exception middleware*). Ova komponenta obrađuje iznimke koje se javljaju tijekom obrade zahtjeva, osiguravajući da se sve pogreške pravilno evidentiraju i obrade. Ona bilježi detalje o iznimkama i vraća odgovarajuće HTTP odgovore klijentu, čime se povećava robusnost aplikacije i olakšava dijagnostika problema.

Druga je komponenta za validaciju koja koristi paket FluentValidation. Ova komponenta posredničkog sloja osigurava da svi ulazni podaci budu pravilno validirani prije nego što stignu do kontrolera. Paket FluentValidation omogućuje definiranje pravila validacije na deklarativan način, što olakšava održavanje i proširivanje validacijskih pravila [10]. Validacija podataka prije obrade smanjuje mogućnost pogrešaka i povećava pouzdanost aplikacije.

Treća komponenta je jednostavna unaprijed ugrađena komponenta za zapisivanje (eng. *logging*). Ova komponenta bilježi informacije o svim HTTP zahtjevima i odgovorima, uključujući vrijeme obrade, statusne kodove i druge relevantne podatke. Logiranje se obavlja u konzolu i log datoteke s pomoću Serilog-a [11], što omogućuje detaljno praćenje rada aplikacije i olakšava dijagnostiku i rješavanje problema.

2.1.5. Mapiranje i DTO-ovi

U aplikaciji PennyPlanner, mapiranje podataka između različitih slojeva aplikacije ključan je dio arhitekture, a za ovaj zadatak koristi se AutoMapper. AutoMapper je popularni .NET paket koji omogućuje automatsko mapiranje objekata, smanjujući potrebu za ručnim kodiranjem i osiguravajući konzistentnost i točnost podataka [12].

Data Transfer Objects (DTO-ovi) su jednostavni objekti koji se koriste za prijenos podataka između slojeva aplikacije. Oni apstrahiraju podatke i poslovnu logiku, čime povećavaju sigurnost i smanjuju zbijenost između slojeva. DTO-ovi su posebno korisni u komunikaciji između klijentskog i poslužiteljskog sloja, gdje je potrebno osigurati prijenos samo relevantnih podataka.

AutoMapper konfiguriran u aplikaciji PennyPlanner pojednostavljuje proces mapiranja između entiteta baze podataka i DTO-ova. Ovo omogućuje automatsku pretvorbu podataka, čime se smanjuje mogućnost pogrešaka i olakšava održavanje koda. Korištenje DTO-ova također poboljšava validaciju podataka, osiguravajući da svi podaci zadovoljavaju potrebne kriterije prije obrade.

2.2. Klijentski sloj

Klijentski sloj aplikacije PennyPlanner odgovoran je za interakciju s korisnikom i pružanje intuitivnog i responzivnog korisničkog sučelja. Ovaj sloj omogućuje korisnicima jednostavno upravljanje osobnim financijama putem preglednog i funkcionalnog sučelja.

2.2.1. Struktura projekta klijentskog sloja

Klijentski sloj aplikacije PennyPlanner organiziran je prema standardnoj Vue.js strukturi koja omogućuje modularnost i jednostavno održavanje. Glavna mapa src sadrži ključne direktorije i datoteke. Direktorij assets sadrži statične resurse poput slika, dok components sadrži Vue komponente koje čine različite dijelove korisničkog sučelja, kao što su AccountsWindow.vue, HomePage.vue, LandingPage.vue, LineChart.vue, GoalsWindow.vue i TransactionsWindow.vue. Direktorij plugins sadrži konfiguracijske datoteke za dodatke kao što su Vuetify i Web Font Loader.

Glavna Vue komponenta App.vue služi kao korijenska komponenta aplikacije, dok datoteka main.js inicijalizira Vue instancu i povezuje sve glavne dijelove aplikacije. Datoteka router.js konfigurira Vue Router za upravljanje navigacijom unutar aplikacije, omogućujući glatko prebacivanje između različitih pogleda. Konfiguracijska datoteka axios.js koristi se za integraciju Axios biblioteke, olakšavajući slanje HTTP zahtjeva poslužiteljskom sloju.

Ostale ključne datoteke uključuju babel.config.js za konfiguraciju Babel-a, jsconfig.json za JavaScript razvojno okruženje, te package.json za upravljanje ovisnostima i metapodacima

projekta. Struktura projekta omogućuje jasno razdvajanje odgovornosti, olakšava rad na različitim dijelovima aplikacije bez izazivanja sukoba ili nejasnoća, te omogućuje lako proširenje i održavanje aplikacije PennyPlanner

2.2.2. Komponente

Komponente su temeljni gradivni blokovi Vue.js aplikacija. U aplikaciji PennyPlanner, komponente se koriste za izradu modularnih, ponovo upotrebljivih dijelova korisničkog sučelja. Svaka komponenta je odgovorna za specifičnu funkcionalnost ili dio sučelja, što omogućuje jasnu separaciju odgovornosti i olakšava održavanje i proširivanje aplikacije.

U direktoriju components nalaze se ključne komponente aplikacije PennyPlanner:

- AccountsWindow.vue: Komponenta koja upravlja prikazom i interakcijom s korisničkim računima. Omogućuje pregled, dodavanje, ažuriranje i brisanje računa.
- HomePage.vue: Početna stranica aplikacije koja pruža osnovne informacije o aplikaciji i omogućuje korisnicima pristup funkcijama prijave i registracije.
- LandingPage.vue: Glavna stranica koja se prikazuje nakon prijave korisnika. Pruža pregled osnovnih financijskih informacija i omogućuje navigaciju do drugih dijelova aplikacije.
- LineChart.vue: Komponenta za prikaz grafova, koja vizualizira financijske podatke korisnika u obliku linijskog grafikona.
- TransactionsWindow.vue: Komponenta koja upravlja transakcijama korisnika, omogućujući pregled, dodavanje, ažuriranje i brisanje transakcija.

Svaka komponenta sastoji se od tri glavna dijela: template, script i style. Template definira HTML strukturu komponente, script sadrži poslovnu logiku i podatke, dok style omogućuje stiliziranje komponente. Ova organizacija olakšava razumijevanje i upravljanje kodom.

Komponente također komuniciraju međusobno putem props i emit događaja, omogućujući fleksibilnu i dinamičnu interakciju između različitih dijelova korisničkog sučelja.

2.2.3. Navigacija

Navigacija unutar aplikacije PennyPlanner implementirana je korištenjem Vue Routera, alata za upravljanje rutama u Vue.js aplikacijama. Vue Router omogućuje definiciju različitih ruta (URL putanja) i povezivanje tih ruta s odgovarajućim komponentama, čime se omogućuje glatka i dinamična navigacija kroz aplikaciju.

Konfiguracija Vue Routera nalazi se u datoteci router.js, koja definira sve potrebne rute za aplikaciju. Svaka ruta specificira putanju i komponentu koja će se prikazati kada korisnik posjeti tu putanju. Na primjer, početna stranica aplikacije povezana je s komponentom HomePage.vue, dok je glavna stranica nakon prijave korisnika povezana s komponentom LandingPage.vue.

2.2.4. Integracija s API-jem

Integracija klijentskog sloja aplikacije PennyPlanner s poslužiteljskim slojem obavlja se putem HTTP zahtjeva koristeći Axios, popularnu biblioteku za upravljanje HTTP zahtjevima u JavaScript-u. Axios omogućuje slanje asinkronih zahtjeva poslužitelju, dohvaćanje podataka i rukovanje odgovorima na jednostavan i efikasan način.

Datoteka axios.js u direktoriju src konfigurira Axios za korištenje unutar aplikacije. Ova konfiguracija može uključivati osnovni URL za API, zadane zaglavlja za zahtjeve i interceptore za obradu zahtjeva i odgovora. Interceptori su korisni za dodavanje tokena za autentifikaciju u svaki zahtjev ili za rukovanje greškama globalno.

Komponente unutar aplikacije koriste Axios za slanje zahtjeva poslužiteljskom API-ju. Na primjer, prilikom kreiranja novog korisničkog računa, komponenta može poslati POST zahtjev s podacima korisnika na odgovarajuću API krajnju točku. Kada poslužitelj odgovori, komponenta može obraditi odgovor i ažurirati stanje aplikacije u skladu s time. Primjer opisanog scenarija nalazi se na slici (Slika 2.5).

```
async register() {
    try {
        await this.$axios.post('/api/User/register', this.registerData);
        this.showRegisterDialog = false;
        this.registerError = null;
        this.loginData = {
            login: this.registerData.username,
            password: this.registerData.password
        };
        await this.login();
    } catch (error) {
        console.error('Registration failed', error);
        try {
            const errorArray = JSON.parse(error.response.data.detail);
            const errorMessages = errorArray.map(error => error.ErrorMessage);
            this.registerError = errorMessages[0];
        }
        catch (e) {
            this.registerError = error.response.data.detail;
        }
    }
}
```

Slika 2.5 Stvaranje API zahtjeva za registraciju

2.2.5. Autentifikacija i autorizacija

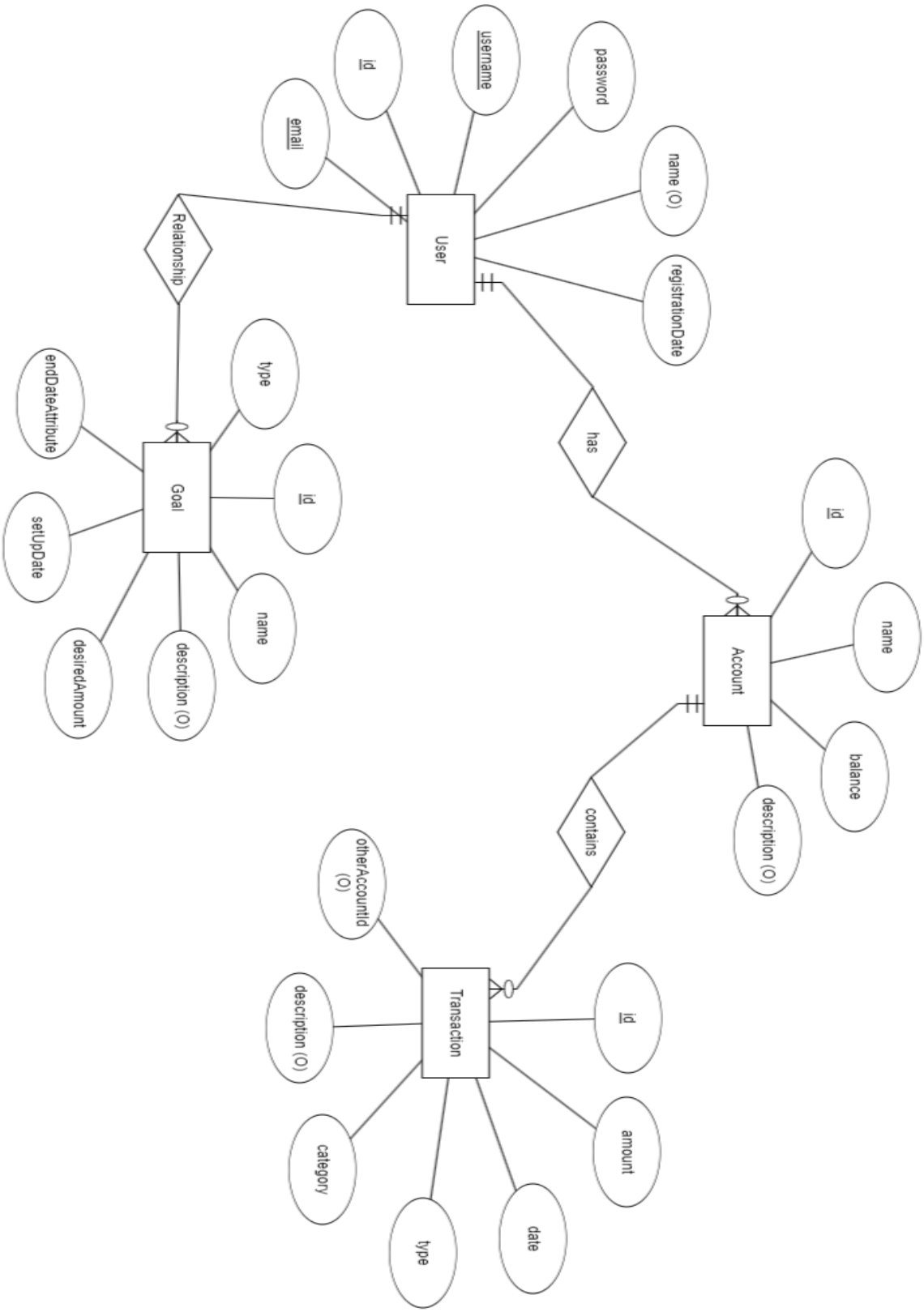
Autentifikacija i autorizacija ključni su dijelovi sigurnosti klijentskog sloja aplikacije PennyPlanner. Ovi procesi osiguravaju da samo ovlašteni korisnici mogu pristupiti određenim dijelovima aplikacije i obavljati određene operacije.

U aplikaciji PennyPlanner, autentifikacija korisnika provodi se putem korisničkog imena/emaila i lozinke. Kada se korisnik uspješno prijavi putem obrasca za prijavu, poslužitelj provjerava unesene podatke kako bi potvrdio identitet korisnika. Ako su podaci ispravni, korisnik je autentificiran i poslužitelj generira JSON Web Token (JWT).

JWT se vraća klijentu i pohranjuje se u lokalnu pohranu (localStorage). Ovaj token sadrži informacije o korisniku i njegovim pravima pristupa te se koristi za autorizaciju svih budućih zahtjeva prema poslužitelju.

Prilikom slanja zahtjeva zaštićenim API krajnjim točkama, klijent uključuje JWT u zaglavlje zahtjeva. Poslužitelj provjerava token kako bi osigurao da je zahtjev poslan od strane autentificiranog korisnika. Ako je token važeći, zahtjev se obrađuje; u suprotnom, korisnik dobiva odgovor o odbijenom pristupu.

2.3. Baza podataka



Slika 2.6 ER dijagram baze podataka

2.3.1. Opis baze podataka

Baza podataka aplikacije PennyPlanner osmišljena je tako da podržava sve ključne funkcionalnosti aplikacije, omogućujući pohranu i upravljanje podacima o korisnicima, računima, transakcijama i financijskim ciljevima. Na slici (Slika 2.6) se prikazuje ER dijagram baze podataka koji obuhvaća četiri glavna entiteta: User, Account, Transaction i Goal.

Entitet User sadrži osnovne informacije o korisniku, uključujući ID, korisničko ime, lozinku, email adresu, ime i datum registracije. Svaki korisnik može imati više računa i ciljeva te obavljati različite transakcije. Entitet Account predstavlja financijski račun korisnika i sadrži atribute poput ID-a, imena računa, stanja i opcionalnog opisa. Svaki račun pripada jednom korisniku i može imati više transakcija. Entitet Transaction evidentira financijske transakcije koje se obavljaju unutar računa. Sadrži ID, iznos, datum, tip, kategoriju, te opcionalne atribute opis i ID drugog računa za prijenos sredstava između računa. Svaka transakcija pripada jednom računu. Entitet Goal predstavlja financijski cilj korisnika, kao što je štednja za određenu svrhu. Sadrži ID, naziv cilja, tip, željeni iznos, datum postavljanja, završni datum i opcionalni opis. Svaki cilj pripada jednom korisniku.

2.3.2. Migracije u Entity Frameworku

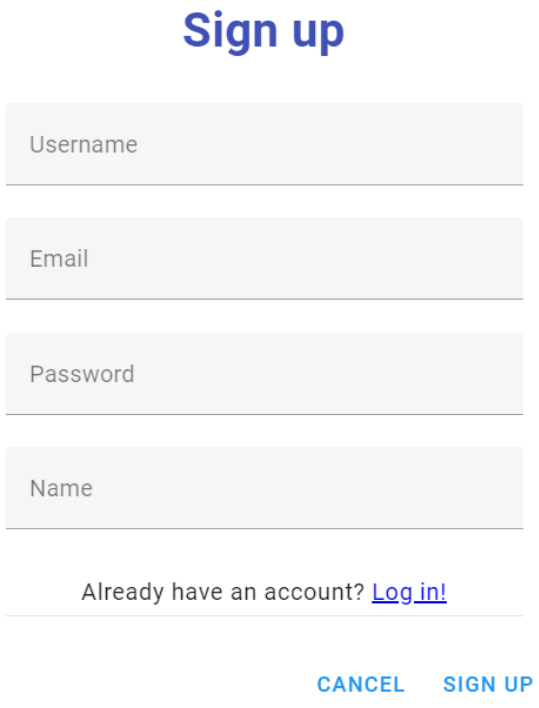
Migracije u Entity Framework Core-u omogućuju upravljanje promjenama u strukturi baze podataka tijekom razvoja aplikacije. Migracije se koriste za dodavanje, mijenjanje ili brisanje tablica i stupaca u bazi podataka bez gubitka podataka. U aplikaciji PennyPlanner migracije se koriste za održavanje sinkronizacije između modela u aplikaciji i stvarne strukture baze podataka. Kada se promijeni model, stvara se nova migracija koja opisuje te promjene. Migracije se zatim primjenjuju na bazu podataka kako bi se ažurirala njena struktura. Proces migracije u Entity Framework Core-u uključuje nekoliko koraka. Kada se model promijeni, kreira se nova migracija pomoću alata kao što je CLI (Command Line Interface) s naredbom dotnet ef migrations add <NazivMigracije>. Ova naredba stvara datoteke koje opisuju promjene u modelu. Nakon što je migracija stvorena, primjenjuje se na bazu podataka pomoću naredbe dotnet ef database update. Ova naredba izvršava sve promjene definirane u migraciji i ažurira strukturu baze podataka.

3. Funkcionalnosti sustava

Aplikacija PennyPlanner nudi raznovrsne funkcionalnosti koje korisnicima omogućuju učinkovito upravljanje osobnim financijama. Ove funkcionalnosti su dizajnirane kako bi korisnicima pružile sve potrebne alate za praćenje prihoda i rashoda, postavljanje financijskih ciljeva te organizaciju računa i transakcija.

3.1. Prijava i registracija korisnika

Prijava i registracija korisnika su osnovne funkcionalnosti koje omogućuju korisnicima pristup aplikaciji i upravljanje svojim financijskim podacima. Prilikom registracije, korisnik unosi osnovne podatke poput korisničkog imena, lozinke, email adrese i imena. Ti podaci se validiraju te se kreira novi korisnički račun. Prijava se obavlja unosom korisničkog imena ili email adrese i lozinke, nakon čega se korisniku dodjeljuje JSON Web Token (JWT) za autentifikaciju. Prijavu ili registraciju moguće je obaviti na odredišnoj stanici (eng. *landing page*) koja uz to pruža i osnovne informacije o aplikaciji. Prozor koji služi za registraciju korisnika prikazan je na slici (Slika 3.1).



Slika 3.1 Odredišna stranica – prozor za registraciju

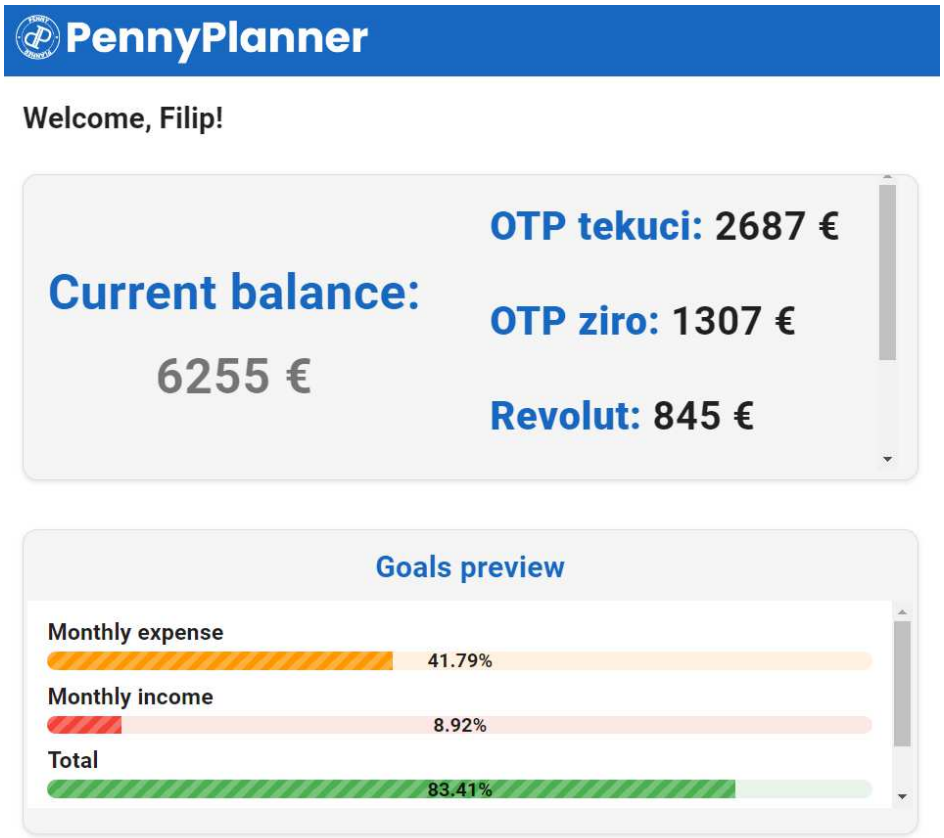
3.2. Korisnički profil

Korisnički profil omogućuje korisnicima pregled i uređivanje osobnih podataka. Korisnici mogu ažurirati svoje ime, korisničko ime, email adresu i lozinku. Promjene se validiraju kako bi se osigurala ispravnost podataka i spriječile greške. Osim toga, korisnički profil prikazuje i datum registracije. Naposljetku, na korisničkom profilu moguće je odjaviti se te izbrisati račun.

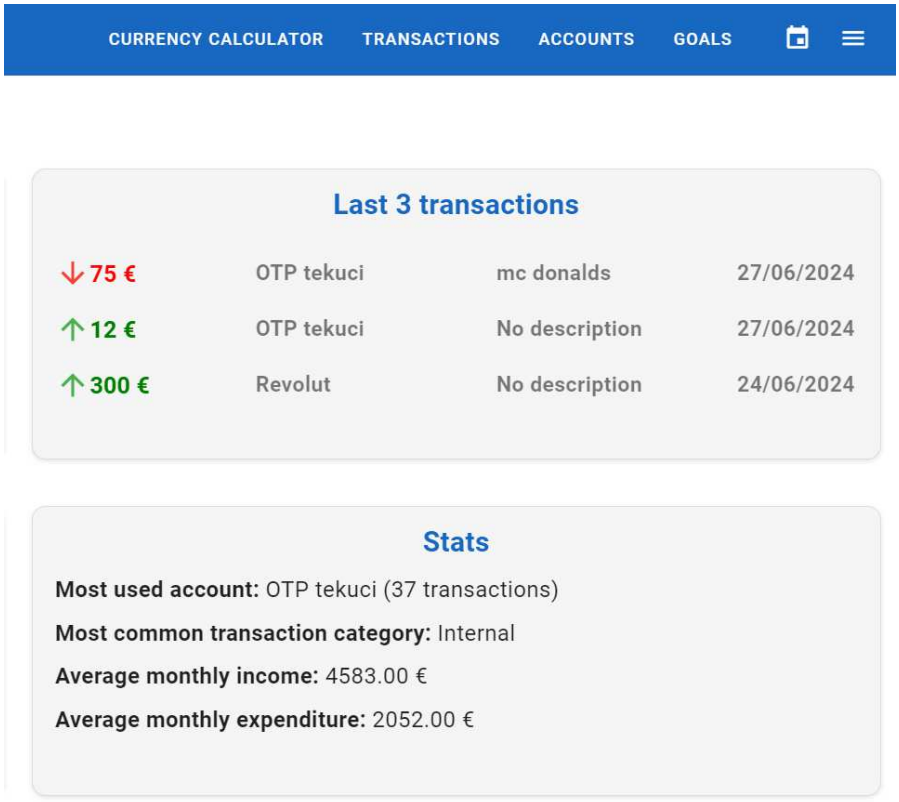
3.3. Početna stranica

Nakon što korisnik obavi prijavu, učitava se početna stranica. Ona na sebi sadrži nadzornu ploču (eng. *dashboard*) koja se sastoji od 4 modula sa najkorisnijim informacijama. Prvi modul prikazuje trenutno stanje pojedinačnih računa te njihov ukupni zbroj. Drugi modul prikazuje posljednje tri transakcije te njihove pojedinosti. Sljedeći modul grafički prikazuje zadane ciljeve, a posljednji modul prikazuje neke od najvažnijih statistika poput računa koji se najčešće koristi, prosječnih mjesečnih primanja i troškova itd.

Na početnoj stranici nalazi se i alatna traka koja omogućuje pristup prozoru za upravljanje računima, prozoru za upravljanje transakcijama, prozoru za upravljanje ciljevima, prozoru za upravljanje korisničkim računom, kalendaru te valutnom kalkulatoru. Prikaz početne stranice moguće je vidjeti na slikama (**Pogreška! Izvor reference nije pronađen.** i Slika 3.3).



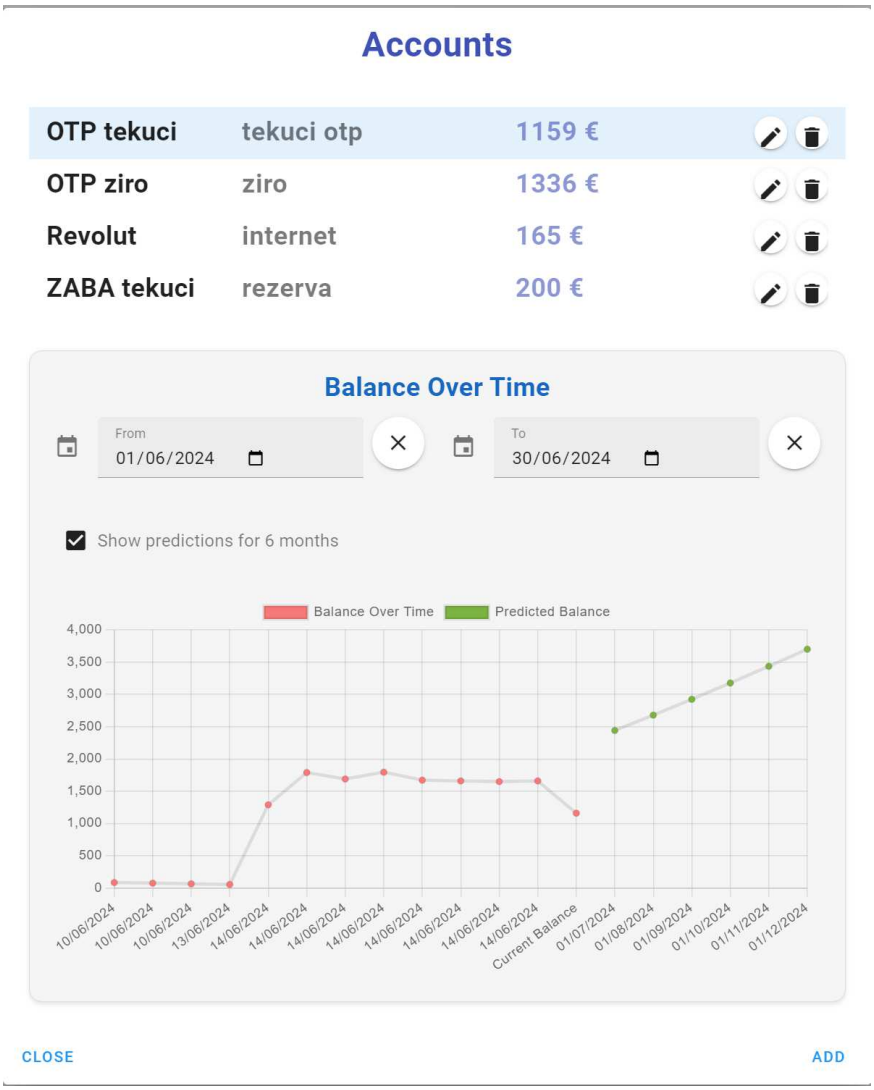
Slika 3.2 Početna stranica – prikaz trenutnog stanja i ciljeva



Slika 3.3 Početna stranica – prikaz transakcija i statistika

3.4. Upravljanje računima

Prozor za upravljanje računima daje nam prikaz svih postojećih računa te informacije za svaki: ime, opis te iznos na računu. Za svaki račun dostupne su opcije brisanja i uređivanja, a kada korisnik pritisne na neki račun, u donjem dijelu prozora pojavljuje se grafički prikaz kretanja iznosa na tom računu. Dostupne su opcije određivanja vremenskog perioda transakcija koje će biti prikazane na grafu te zastavica koja uključuje predikciju kretanja stana računa u sljedećih 6 mjeseci, a predikcija se izračunava na temelju dostupnih transakcija. Prozor je vidljiv na slici (Slika 3.4). Na dnu se nalaze tipke za dodavanje novog računa te zatvaranja cijelog prozora.



Slika 3.4 Prozor za upravljanje računima

3.4.1. Predikcija stanja računa

Predikcija stanja računa u aplikaciji PennyPlanner ostvarena je korištenjem polinomske regresije drugog stupnja i ponderiranog pomičnog prosjeka. Proces započinje tako da se ulazni podaci o prethodnim stanjima računa prolaze kroz funkciju za izračun ponderiranog pomičnog prosjeka. Ova metoda smanjuje fluktuacije u podacima i stvara smirenu seriju podataka, pri čemu se veći značaj daje novijim podacima.

Nakon smirivanja podataka, pripremaju se podaci za polinomsku regresiju. Indeksi podataka mapiraju se na x vrijednosti, a smireni podaci na y vrijednosti. Matrica X popunjava se snagama indeksa od 0 do 2, što omogućava prilagodbu podacima pomoću polinoma drugog stupnja.

Sljedeći korak uključuje korištenje Gaussove eliminacije za izračunavanje koeficijenata polinoma. Funkcija za Gaussovu eliminaciju transformira matricu u gornji trokutasti oblik i zatim koristi unatrag supstituciju kako bi se dobili koeficijenti polinoma.

Kada su koeficijenti polinoma izračunati, koriste se za predviđanje budućih stanja računa. Vrijednosti x za buduće vremenske periode (indeksi nakon trenutne duljine podataka) unose se u polinomski izraz kako bi se dobile predviđene y vrijednosti, odnosno buduća stanja računa. Predikcije se vrše za narednih šest mjeseci. Ovim postupkom aplikacija može predvidjeti buduća stanja računa na temelju povijesnih podataka, omogućujući korisnicima bolje planiranje i upravljanje svojim financijama. Najvažniji isječak koda za predikciju prikazan je na slici (Slika 3.5).

```
getPredictedBalances(data) {
    const predictions = [];
    const numPredictions = 6;
    const degree = 2;

    const smoothedData = this.weightedMovingAverage(data, 5);

    const x = smoothedData.map((_, index) => index);
    const y = smoothedData;

    const n = y.length;
    const X = [];
    const Y = y;

    for (let i = 0; i < n; i++) {
        X[i] = [];
        for (let j = 0; j <= degree; j++) {
            X[i].push(Math.pow(x[i], j));
        }
    }

    const XT = X[0].map((_, colIndex) => X.map(row => row[colIndex]));
    const XTX = XT.map(row => row.map((_, colIndex) => row.reduce((sum, cell, i) => sum + cell * X[i][colIndex], 0)));
    const XTY = XT.map(row => row.reduce((sum, cell, i) => sum + cell * Y[i], 0));
    const B = this.gaussianElimination(XTX, XTY);

    for (let i = 1; i <= numPredictions; i++) {
        const predictedX = x.length + i;
        let predictedY = 0;
        for (let j = 0; j <= degree; j++) {
            predictedY += B[j] * Math.pow(predictedX, j);
        }
        predictions.push(Math.round(predictedY));
    }

    return predictions;
},
```

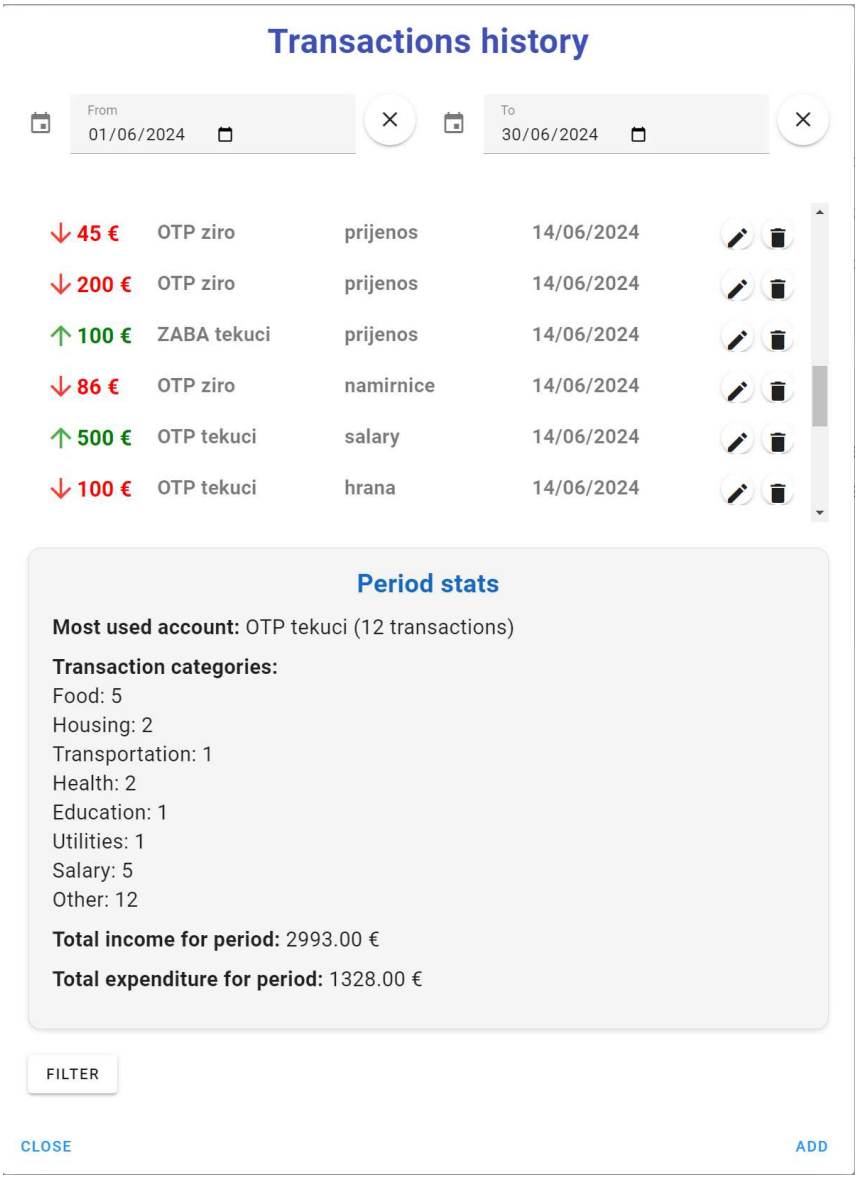
Slika 3.5 Metoda getPredictedBalances

3.5. Upravljanje transakcijama

Prozor za upravljanje transakcijama u gornjoj polovici prikazuje popis svih transakcija i njihove najbitnije informacije, a te transakcije moguće je filtrirati vremenski te po ostalim parametrima koji definiraju transakciju – račun, trošak, prihod ili predložak te kategorija transakcije npr. hrana, stanarina, zdravlje, potrepštine, zabava, plaća itd. Svaku transakciju moguće je obrisati, a u slučaju da se radi o predlošku transakcije, moguće je i aktivirati taj predložak.

U donjoj polovici nalaze se statistike za označeni period, a to su: najčešće korišteni račun, raspodjela transakcija po kategorijama, kumulativni prihod za označeni period te kumulativni trošak.

Na dnu prozora nalaze se gumbi za dodavanje nove transakcije te zatvaranje prozora, a sam prozor prikazan je na slici (Slika 3.6).



Slika 3.6 Prozor za upravljanje transakcijama

Kod dodavanja nove transakcije, između ostaloga dostupna je i opcija „other account“ koja omogućava interne prijenose sredstava između računa istog korisnika. Kod brisanja neke transakcije ponuđena je opcija „restore balance“. Ako korisnik odabere tu opciju, prilikom brisanja transakcije sredstva na računu bit će postavljanje u stanje kakvo bi bilo da se ta transakcija nikada nije ni unijela.

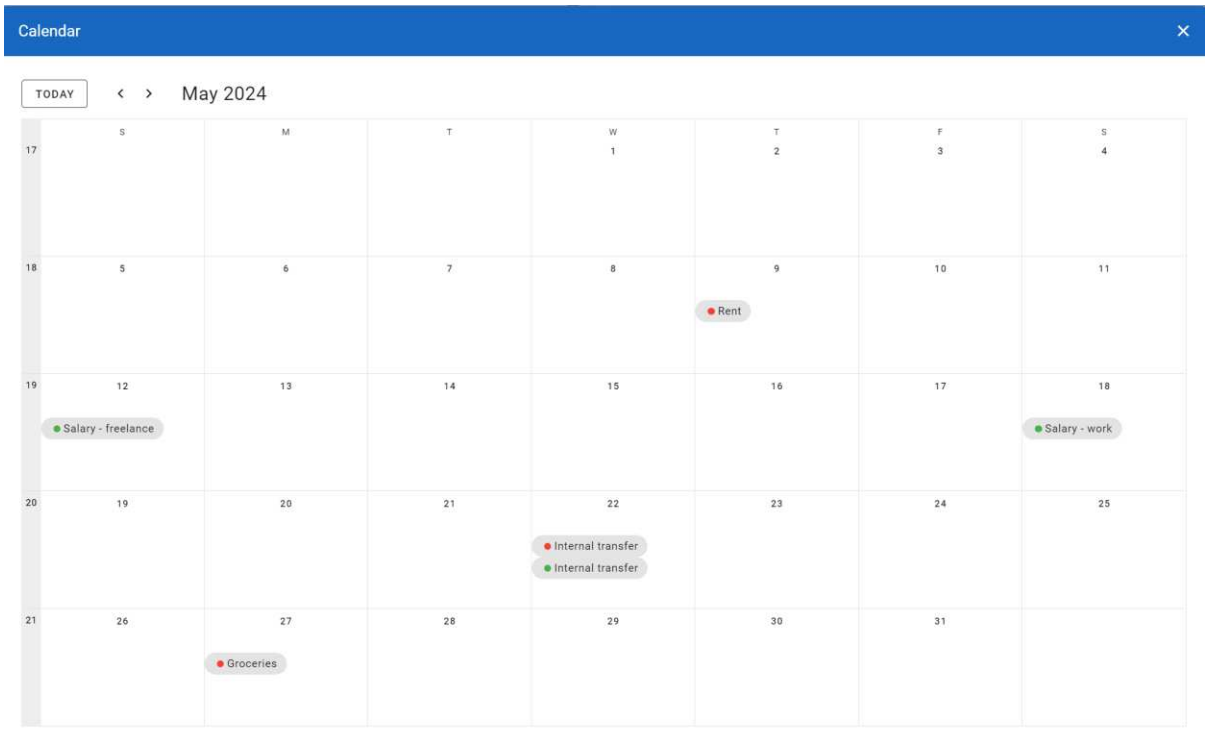
3.6. Upravljanje ciljevima

Prozor za upravljanje ciljevima prikazuje detaljnije informacije o ciljevima koji se mogu vidjeti na početnoj stranici (**Pogreška! Izvor reference nije pronađen.**). Za svaki cilj prikazan je naziv, grafički prikaz sličan onome na početnoj stranici, trenutno novčano stanje vezano za taj cilj u usporedbi s ciljnim stanjem, vrijeme preostalo za ispunjenje cilja te opis cilja. Slično kao i kod ostalih prozora za upravljanje entitetima, dostupne su opcije za dodavanje novog cilja te za uređivanje i brisanje bilo kojeg od dostupnih ciljeva. Postoje 4 vrste ciljeva koje se dodatno još dijele u dvije kategorije. Prve dvije vrste „ukupan novac“ te „novac na jednom računu“ imaju mogućnost da korisnik sam odredi vrijeme da se takav cilj ispuni, dok je kod sljedeće dvije vrste „mjesečni prihod“ te „mjesečni trošak“ vremenski period automatski postavljen na mjesec dana. Prozor je vidljiv na slici (Slika 3.7).

Slika 3.7 Prozor za upravljanje ciljevima

3.7. Ostali prozori

Osim dosad opisanih prozora, postoje još valutni kalkulator te kalendar. Valutni kalkulator omogućuje brzu i jednostavnu pretvorbu iznosa između 10 najkorištenijih valuta na svijetu, a trenutne vrijednosti tečaja dohvaćaju se sa stranice exchangerate-api.com. Na kalendaru je moguće vidjeti uređeni prikaz svih transakcija te datuma kada istječu pojedini ciljevi. Prihodne transakcije označene su zelenom, a rashodne crvenom bojom, dok su krajnji datumi ciljeva označeni plavom bojom. Kalendar je prikazan na slici (Slika 3.8).



Slika 3.8 Kalendar

4. Zaključak

U vremenu kada su financijski izazovi svakodnevna stvarnost, mnogi osjećaju potrebu za boljim upravljanjem svojim osobnim financijama. Uz brojne aplikacije koje svakodnevno koriste za trošenje i primanje novaca, postaje otežano pratiti stanje na svim tim platformama. Zbog toga je potreba za jedinstvenom aplikacijom koja omogućuje jednostavno i efikasno upravljanje financijama postala neophodna.

PennyPlanner je razvijena kako bi pomogla korisnicima u organizaciji i planiranju njihovih financija. Aplikacija omogućuje kreiranje i upravljanje različitim računima, bilježenje i kategorizaciju transakcija te postavljanje financijskih ciljeva. Kroz intuitivno korisničko sučelje, korisnici mogu jednostavno pratiti svoje financijske tokove, analizirati troškove i prihode te planirati buduće financijske aktivnosti.

Posebno je značajna funkcionalnost predikcije stanja računa, koja korisnicima omogućuje bolje planiranje i donošenje informiranih odluka o svojoj potrošnji i štednji. Ova funkcionalnost, zajedno s mogućnošću detaljnog praćenja i upravljanja transakcijama, čini PennyPlanner moćnim alatom za svakoga tko želi imati kontrolu nad svojim financijama.

PennyPlanner također doprinosi povećanju financijske pismenosti korisnika, pružajući im alate za efikasno upravljanje osobnim financijama i postizanje dugoročnih financijskih ciljeva. Korištenje ove aplikacije može smanjiti financijski stres i povećati sigurnost, omogućujući korisnicima da bolje razumiju i upravljaju svojim financijama u svakodnevnom životu.

Literatura

[1] *Overview of ASP.NET Core* (2024, lipanj). Poveznica: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0; pristupljeno: 12. lipnja 2024.

[2] *What is a RESTful API?* (2024.). Poveznica: https://aws.amazon.com/what-is/restful-api; pristupljeno: 12. lipnja 2024.

[3] *HTTP*. Poveznica: https://hr.wikipedia.org/wiki/HTTP; pristupljeno: 12. lipnja 2024.

[4] *What is SQLite?* (2021, prosinac). Poveznica: https://builtin.com/data-science/sqlite; pristupljeno: 12. lipnja 2024.

[5] *Entity Framework Core* (2021, svibanj). Poveznica: https://learn.microsoft.com/en-us/ef/core; pristupljeno: 12. lipnja 2024.

[6] *Introduction to Vue3* (2022, listopad). Poveznica: https://code.pieces.app/blog/getting-started-with-vuejs-introduction-to-vue-3; pristupljeno: 13. lipnja 2024.

[7] *Get started with Vuetify 3* (2024, svibanj). Poveznica: https://vuetifyjs.com/en/getting-started/installation; pristupljeno: 13. lipnja 2024.

[8] *Controller-Service-Repository pattern* (2021, kolovoz). Poveznica: https://tom-collings.medium.com/controller-service-repository-16e29a4684e5; pristupljeno: 13. lipnja 2024.

[9] *Desing Pattern Explained – Dependency injection with Code Examples* (2024, ožujak). Poveznica: https://stackify.com/dependency-injection/; pristupljeno: 14. lipnja 2024.

[10] *FluentValidation.* Poveznica: https://docs.fluentvalidation.net/en/latest/; pristupljeno: 14. lipnja 2024.

[11] *Flexible, structured events - log file convenience*. Poveznica: https://serilog.net/; pristupljeno: 14. lipnja 2024.

[12] *Automapper*. Poveznica: https://automapper.org/; pristupljeno: 14. lipnja 2024.