WEB APLIKACIJA ZA TRGOVANJE BITCOINOM

Prusac, Luka

Undergraduate thesis / Završni rad

2021

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

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:228:390169

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Repository of University Department of Professional Studies

zir.nsk.hr

UNIVERSITY OF SPLIT

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

LUKA PRUSAC

Z A V R Š N I R A D

WEB APLIKACIJA ZA TRGOVANJE BITCOINOM

Split, rujan 2021.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

Predmet: Programski alati na Unix računalima

Z A V R Š N I R A D

Kandidat: Luka Prusac

Naslov rada: Web aplikacija za trgovanje bitcoinom

Mentor: Nikola Grgić, viši predavač

Split, rujan 2021.

Sadržaj

Sažetak *1*

1 Uvod *2*

2 Korištene tehnologije *3*

3 Izrada poslužitelja *4*

4 Izrada korisničkog sučelja *55*

5 Prikaz korisničkog sučelja *63*

6 Zaključak *68*

7 Literatura *69*

Sažetak

Web aplikacija za trgovanje Bitcoinom

U ovom radu prikazana je izrada web aplikacije za simulaciju razmjene kriptovalute Bitcoin. Ova aplikacija je razvijena s ciljem da omogući korisniku testiranje trgovanja kriptovaluta, a specifično se koristi kriptovaluta Bitcoin. Aplikacija se sastoji od korisničkog sučelja (engl. *frontend*) te poslužiteljskog dijela (engl. *backend*). Korisničko sučenje web aplikacije služi za prikaz i upravljanje resursima te razmjenu valuta. Pruža korisnicima mogućnost izrade novog računa, podizanje novih narudžbi te pregled povijesti narudžbi i transakcija. Poslužiteljski dio aplikacije se koristi za pohranu podataka o korisnicima i njihovim narudžbama kao i oblikovanje podataka iz baze za korisničko sučelje. Glavna funkcija poslužiteljskog dijela je spajanje korisničkih narudžbi i obrade transakcija među narudžbama.

Ključne riječi: *kriptovaluta, bitcoin, mjenjačnica, simulacija*

Summary

Web application for exchanging Bitcoin

This document shows the creation of the web application for simulation based exchange of Bitcoin cryptocurrency. This application is developed with the goal of enabling the user to test the exchange of Bitcoin. The application consists of two parts; user interface and backend server. The user interface is used for resource management and the exchange of currency via the browser application. It enables users to create new accounts, create new orders or view the order's and transaction's history. The backend server stores data as well as does data modification for requests from the user interface. The main function of the backend server is to match orders and conduct transactions between users.

Keywords: *cryptocurrency, bitcoin, exchange, simulation*

1. Uvod

Zadatak ovog završnog rada je izrada platforme za razmjenu kriptovalute Bitcoin. Iako aplikacija za trgovinu Bitcoinom (Bitcoin mjenjačnica) funkcionira kao izolirana simulacija trgovanja te korisnicima ne pruža mogućnost spajanja s bankovnim računom ili Bitcoin novčanikom, programski kôd poslužiteljskog dijela aplikacije je dovoljno modularno napravljen da se spomenuto spajanje računa može postići nadogradnjom. Taj proces nadogradnje će naknadno biti detaljnije objašnjen.

Aplikacija se sastoji od dva dijela, poslužiteljski dio (engl. *backend*) te korisničko sučelje (engl. *frontend*). Kôd za poslužiteljski dio je pisan u jeziku C# koristeći razvojni okvir .NET Core te Entity Framework koji služi kao alat za lakše rukovanje modelima i bazom podataka. Korisničko sučelje je napisano u jeziku Javascript koristeći biblioteku React za izgradnju dinamične web stranice.

Poslužiteljski dio je izrađen u integriranom razvojnom okruženju Microsoft Visual Studio 2019. Sve dodatne alate i "third party" biblioteke se preuzimaju preko internog NuGet package managera koji to sve jednostavno integrira u projekt. Uz to, Visual Studio uz pomoć Entity Frameworka se brine za izgradnju te pristup bazi podataka pomoću konfigurabilnih parametara unutar projekta. Za izgradnju korisničkog sučelja koristio se Microsoft Visual Studio Code (skraćeno VS Code). VS Code za razliku od Visual Studio 2019 nije integrirano okruženje za razvoj aplikacija, ali je moćan uređivač teksta koji ima mogućnosti instalacije raznih potrebnih dodataka kako bi se olakšalo pisanje koda. Uz VS Code bitno je spomenuti i Node.js koji ima integralnu ulogu u razvoju frontend aplikacije. Uz pomoć Node.js-a i njemu pripadajućeg Node package managera (skraćeno npm) može se preuzeti čitava biblioteka React te svi drugi potrebni alati i dodaci (sa skriptom `npm install`), a sa skriptom `npm run` se može pokrenuti razvojni server kako bi se mogla korisnicima uslužiti stranicu.

2. Korištene tehnologije

Glavne korištene tehnologije i razvojna okruženja za poslužiteljski dio aplikacije su vezane za integrirano razvojno okruženje Microsoft Visual Studio. Od programskog jezika C# do razvojnog okruženja .NET Core, Visual Studio pruža olakšan razvoj poslužiteljih aplikacija. Programski jezik C# je objektno orijentirani jezik razvijen za potrebe razvojnog okruženja .NET [1]. Rađen po uzoru na programske jezike C i Java, mogu se lako primjetiti sličnosti između navedenih jezika. Razvojno okruženje .NET Core pruža korisnicima skup standarnih biblioteka za izradu sveobuhvatnih web servisa. Dodatni alati za razvoj poslužiteljskog dijela aplikacije su Entity Framework i NuGet package manager. Razvojno okruženje Entity Framework pruža korisnicima pristup internim metodama i funkcionalnostima za rad s bazom podataka uz pomoć sintakse Language Integrated Query (Linq) koja omogućuje stvaranje upita SQL iz programskog jezika [2] dok je NuGet package manager odgovoran za instalaciju i praćenje "third party" biblioteka u programu.

Za korisničko sučelje koristila se biblioteka React u paru s okruženjem za izvođenje Javascript koda Node.js. Biblioteka React se koristi za izradu reaktivnih jednostraničnih (engl. *single-page*) web aplikacija. Za pisanje koda elemenata stranice unutar React komponenti koristi se posebna sintaksa JavaScript Syntax Extension (skraćeno JSX) [3]. JSX je zapravo nadogradnja standardne sintakse JavaScript koja omogućuje pisanje HTML elemenata direktno unutar JavaScript koda. Node.js je potreban za pokretanje i obradu JavaScript koda izvan internet preglednika [4]. Također omogućuje i podizanje poslužitelja za korisničko sučelje kako bi se moglo pristupiti web stranici preko URL-a. Node.js uključuje i Node package manager koji ima sličnu funkciju kao i NuGet package manager za pripremu i instalaciju potrebnih "third party" biblioteka. Npm također uključuje i CLI komande (npm skripte) koje će se koristiti za pokretanje korisničkog sučelja, odnosno njegovog poslužitelja. Kao nadopuna biblioteci React, koristi se i biblioteka Material-UI koja sadrži neke već gotove i stilizirane React komponente po uzoru na najosnovnije HTML elemente kako bi se postigao uniformni izgled web stranice. Za vizualno predstavljanje podataka i prikaz grafova na korisničkom sučelju koristila se biblioteka Apache Echarts, točnije biblioteka `echarts-for-react`.

3. Izrada poslužitelja

Izrada poslužiteljskog dijela aplikacije se odvijala prvenstveno unutar programa Microsoft Visual Studio 2019. Unutar Visual Studia mogu se pronaći već spomenuti alati kao što je NuGet package manager kao i .NET Core alati potrebni prilikom pisanja i izvođenja kôda.

S NuGet treba skinuti nekoličinu biblioteka kao što su:

- `AspNetCore.Authentication`
- `AspNetCore.Mvc.NewtonsoftJson`
- `EntityFrameworkCore.SqlServer`
- `EntityFrameworkCore.Tools`

`AspNetCore.Authentication` se koristi kako bi se omogućilo jednostavno korištenje autentikacijskih i autorizacijskih procesa za korisnike kao što su generiranje i validacija tokena te pristup određenim pristupnim točkama (engl. *endpoints*) s obzirom na ulogu korisnika, među ostalim funckijama.

`AspNetCore.Mvc.NewtonsoftJson` se koristi kao pomoćni alat prilikom generiranja odgovora od servera u određenim situacijama. Biblioteka sadrži razne alate za formatiranje JSON datoteka.

`EntityFrameworkCore.SqlServer` biblioteka omogućava korištenje Entity Framework Core-a sa Microsoft SQL serverom. Omogućava i modifikacije podataka prilikom migracija SQL baze.

`EntityFrameworkCore.Tools` pružaju CLI komande za upravljanje bazom podataka. Te komande uključuju, među ostalima stvaranje migracija, primjenu migracija, podizanje baze prema već prisutnim modelima ili generiranje modela prema već prisutnoj bazi podataka, itd. U slučaju ove aplikacije, prvo će se ispisati modeli podataka prema kojima će se naknadno raditi migracije i baza podataka (*Code first* pristup).

Uz to, omogućen je pristup i čitavom `System` namespace-u i njegovim metodama i svojstvima gdje se nalaze i razne Linq funkcije (`System.Linq`) potrebne za manipulaciju

podatcima iz baze podataka (zamjena za SQL pozive) kao i pristup asinkronim metodama (System.Threading), i tako dalje.

Počevši od same strukture direktorija za projekt, mogu se istaknuti bitni poddirektoriji i datoteke:

- `Models/`
  - `UserAccount.cs`
  - `Order.cs`
  - `Transaction.cs`
  - `OrderTransaction.cs`
- `DbContext/`
  - `ExchangeContext.cs`
- `Controllers/`
  - `HomeController.cs`
  - `UserAccountsController.cs`
  - `OrdersController.cs`
  - `TransactionsController.cs`
  - `OAuth.cs`

I dvije bitne datoteke:

- `Startup.cs`
- `Program.cs`

Navedene datoteke i direktoriji su najznačajniji za spomenuti te osim `Startup` i `Program` datoteka sve druge datoteke unutar navedenih direktorija će korisnik sam trebati napraviti, ali definitivno nisu jedine prisutne te .NET projekt generira nekoličinu drugih direktorija i datoteka potrebnih za njegovu funkcionalnost, ali oni neće biti obrađeni u sklopu ovog dokumenta.

Direktorij `Models` sadrži datoteke s opisom modela podataka kako bi trebali biti spremljeni u bazi podataka (tablice).

- Model `UserAccount` sa svojim svojstvima (Ispis 1):

```
public class UserAccount
{
        public Guid ID { get; set; }
        public string Name { get; set; }
        public string Password { get; set; }
        public long AccountBalanceUSD { get; set; }
        public long OrderBalanceUSD { get; set; }
        public long AccountBalanceBTC { get; set; }
        public long OrderBalanceBTC { get; set; }
        public string Role { get; set; }
        public bool ActiveStatus { get; set; }
        [JsonIgnore]
        public virtual ICollection<Order> Orders { get; set; }
}
```

Ispis 1: Primjer klase UserAccount

Primjer objekta klase `UserAccount` (Ispis 2):

```
ID: a30611b1-c420-4282-95a5-08d91d31c018
Name: "Korisnik"
Password: "1234567890"
AccountBalanceUSD: 10000        ($100.00)
OrderBalanceUSD: 0
AccountBalanceBTC: 100000000    (฿1)
OrderBalanceBTC: 5000000        (฿0.05)
Role: "User"
ActiveStatus: true              (Active)
```

Ispis 2: primjer objekta UserAccounts

Virtualna kolekcija `Orders` je skup entiteta tipa `Order` koji su direktno povezani s `UserAccount` entitetom. Taj atribut služi Entity Frameworku da zna u kakvom su odnosu dvije tablice. U ovom slučaju, kako entitet `UserAccount` sadrži kolekciju entiteta `Order`, radi se o 1-N vezi (iako bi trebalo ustvrditi da i `Order` entitet nema kolekciju entiteta `UserAccount`, ali u ovom slučaju to je istina). Kada se bude spominjao model `Order`, tada će biti prikazan virtualni atribut `UserAccount`.

Atribut `[JsonIgnore]` se koristi kod serijalizacije entiteta (odnosno označava atribut koji se neće serijalizirati) te služi kako bi se izbjegao *self-referencing loop* kod dohvaćanja kolekcije `Orders` jer kao što je spomenuto, svaki entitet `Order` ima svoje "virtualno" svojstvo `UserAccount` koje sadrži referencu na svoju kolekciju narudžbi, referencirajući pritom sam sebe. Kako ne bi u nedogled referencirali `Orders` preko `UserAccount` i suprotno, `JsonIgnore` atribut je upotrebljen. Zbog tog razloga svaki model će imati prisutan atribut `[JsonIgnore]` za svoja virtualna svojstva.

`Column`, `Display` i `StringLength` su drugi opisni atributi koji određuju dodatne opcije spremanja u bazu podataka. `[Column]` atribut označava 'ID' stupca, `[Display]` atribut označava ime stupca u tablici a `[StringLength]` daje dodatne restrikcije na oblik podatka koji se sprema (u ovom slučaju, za primjer stupca `Name` će to biti niz slova maksimalne dužine 50 znakova, ali minimalne dužine 3 znaka).

Da bi ovaj model podataka proširili na pristup stvarnim korisnikovim računima, trebalo bi dodati novi model/datoteku koja će sadržavati informacije o računima korisnika. Taj novi model zatim treba spojiti s trenutnim modelom `UserAccount` kao što je napravljeno s vezom između modela `UserAccount` i `Order`. Tada bi svojstvo `AccountBalance` odražavalo samo zadnje stanje korisnikovih stvarnih računa, ali onda se može i izostaviti u tom slučaju.

- Model `Order` sa svojim svojstvima (Ispis 3):

```
public class Order
    {
        public Guid ID { get; set; }
        [ForeignKey("UserAccount")]
        public Guid UserID { get; set; }
        public long OrderAmount { get; set; }
        public long OrderRemainingSize { get; set; }
        public long OrderPrice { get; set; }
        public DateTime OrderDate { get; set; }
        public bool OrderOpenStatus { get; set; }
        public string OrderType{ get; set; }
        [JsonIgnore]
        public virtual UserAccount UserAccount { get; set; }
        [JsonIgnore]
        public virtual ICollection<OrderTransaction> Transactions{
           get; set; }
     }
```

Ispis 3: Primjer klase Order

Primjer objekta klase `Order` (Ispis 4):

```
ID: 256d4454-266f-4bdf-b936-08d91d36476d

UserID: a30611b1-c420-4282-95a5-08d91d31c018

OrderAmount: 100000000                    // Ƀ1

OrderRemainingSize: 50000000              // Ƀ0.5

OrderPrice: 3500000                       //
$35,000.00

OrderDate: 01/01/2021 11:11:11

OrderOpenStatus: true                     // Open

OrderType: "Buy Limit Order"
```

Ispis 4: Primjer objekta Order

U modelu `Order` se nalazi više virtualnih svojstava zbog relacije između ostalih tablica. Virtualno svojstvo `UserAccount` zajedno sa svojstvom `UserID` čine poveznicu s pripadajućim entitetom `UserAccount` koji je otvorio tu narudžbu. Atribut `[ForeignKey]` označava svojstvo `UserID` kao strani ključ u tablici (u vezi prema navedenoj tablici `UserAccount`). Istovremeno svi entiteti `Order` s pripadajućim `UserID` se nalaze u virtualnoj kolekciji `Orders` u entitetu `UserAccount`. Virtualna kolekcija `Transactions` (ovog puta drugačijeg tipa, `OrderTransaction`) označava listu transakcija u kojima je sudjelovala trenutna narudžba. Oba svojstva imaju `[JsonIgnore]` atribut zbog deserijalizacije tih podataka prilikom ispisa. Razlog je naveden u paragrafu za UserAccount model te će se kod daljnjih modela opis tog atributa totalno ignorirati, ali njegova svrha je uvijek ista i jedinstvena. Osim navedenih svojstava, klasa `Order` ima i dvije metode (Ispis 5) iako se rijeđe koriste:

```
public bool TypeEquals(string type)
{
      var orderType = this.OrderType.Split(" ").ElementAt(1);
      return orderType.ToUpper() == type.ToUpper();
}
 public bool OrderVariationType(string type)
{
      var orderType = this.OrderType.Split(" ").ElementAt(0);
     return orderType.ToUpper() == type.ToUpper();
}
```

Ispis 5: Primjer koda pomoćnih metoda klase Order

`TypeEquals` metoda vraća tip narudžbe (*Buy* ili *Sell* narudžba) a `OrderVariationType` vraća varijaciju tipa narudžbe (*Limit* ili *Market*). Kako je polje 'tip narudžbe' spremljen kao jedan pojam tipa `string`, preko ovih metoda mogu se izvući koncizne informacije iz svojstva `OrderType`.

- Model `Transaction` sa svojim svojstvima (Ispis 6):

```
public class Transaction
{
        public Guid ID { get; set; }
 [DisplayFormat(DataFormatString = "{0:HH:mm:ss, dd-MM-yyyy}"]
        DateTime Date { get; set; }
        public long TradedBitcoin { get; set; }
        public long TradedPrice { get; set; }
        [JsonIgnore]
        public virtual ICollection<OrderTransaction> Orders {
get; set; }
        [JsonIgnore]
        public virtual ICollection<UserTransactions> Users {
get; set; }
}
```

Ispis 6: Primjer klase Transaction.cs

Primjer objekta klase `Transaction` (Ispis 7):

```
ID: 336425bd-1cca-4a78-b5db-08d92a076bfb

Date: 02/02/2021 22:22:22

TradedBitcoin: 50000000        (฿0.5)

TradedPrice: 3500000           ($35,000.00)
```

Ispis 7: Primjer objekta Transaction

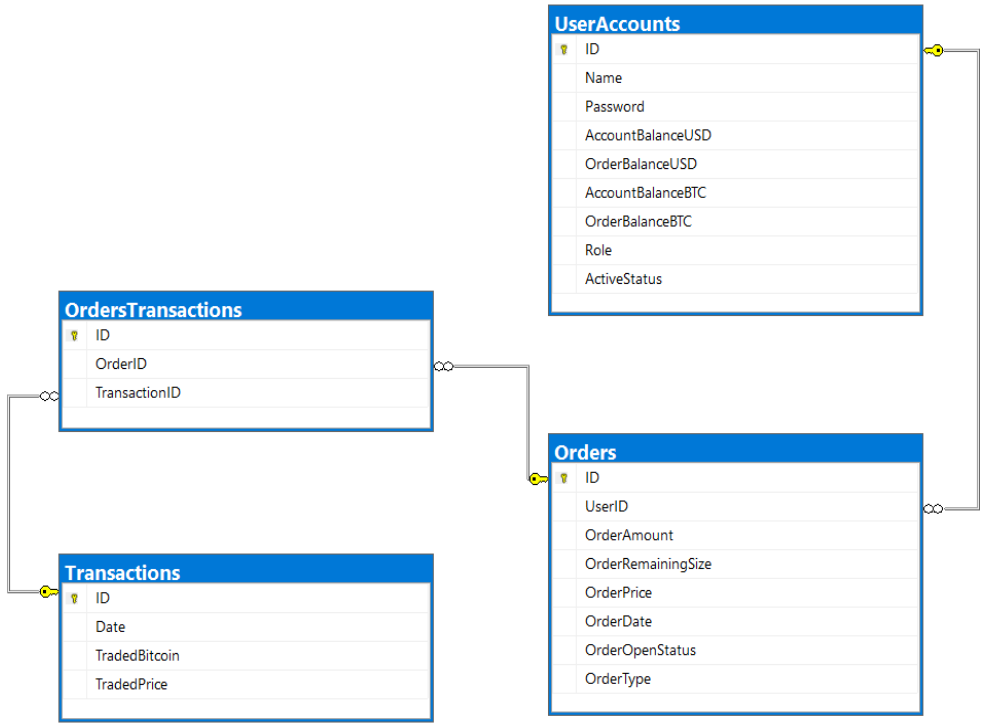
Atribut `[DisplayFormat]` koji određuje format ispisa retka u tablici. Dodan je samo zbog estetskih razloga i lakšeg čitanja podataka. Važnija od toga je pojava nove virtualne kolekcije `Orders` (tipa `OrderTransaction`). Zbog prisutnosti virtualne kolekcije `Orders`, iste kolekcije koja se nalazila u modelu `Order`, se može zaključiti da su ove dvije tablice, `Order` i `Transaction` u relaciji N-N te će biti potrebna još jedna tablica (`OrderTransaction`) kako bi upotpunili vezu između entiteta.

- Model `OrderTransaction` sa svojim svojstvima (Ispis 8):

```
public class OrderTransaction
{
    public Guid ID { get; set; }
    [ForeignKey("Order")]
    public Guid OrderID { get; set; }
    [JsonIgnore]
    public Order Order { get; set; }
    [ForeignKey("Transaction")]
    public Guid TransactionID { get; set; }
    [JsonIgnore]
    public Transaction Transaction { get; set; }
}
```

Ispis 8: Primjer klase OrderTransaction.cs

Ova međutablica je realizirana na način da sadrži identifikatore za oba podudarajuća entiteta (`Order` i `Transaction`). Njihova pripadajuća svojstva `Order` *Order* i `Transaction` *Transaction* ovog puta nisu virtualna jer pripadaju trenutnoj međutablici, ali njihova funkcionalnost je više manje nepromijenjena od virtualnih svojstava prethodnih modela. Prikaz dijagrama baze podataka sa svim modelima (Slika 1):



Slika 1: Dijagram modela podataka.cs

Kada je čitav model podataka prisutan, može se dodati pripadajući `DbContext` za postojeću strukturu podataka. Unutar direktorija `DBContext` i same datoteke `ExchangeContext`, treba definirati setove konteksta za svaki model kako bi Entity Framework znao spajati naše objekte i klase s podacima iz baze podataka. Sve to treba biti unutar klase `ExchangeContext` koja nasljeđuje klasu `DbContext` te tako dobiva pristup nekim metodama i svojstvima koji su potrebni za izgraditi potpuni kontekst baze podataka. Prije nego se definiraju potrebni setovi, dodatno će se ustvrditi veze između prethodno definiranih entiteta s metodom `OnModelCreating`. U njoj će se dodatno definirati sve veze među modelima (veze koje su već navedene preko svojstava kao što su strani ključevi i druga virtualna svojstva). Međutim, ovim korakom će se ukloniti sve potencijalne nedoumice i

uvesti potrebne redundancije u integritet podataka. Metoda `OnModelCreating` (Ispis 9) je naslijeđena od klase `DbContext`.

```
protected override void OnModelCreating(ModelBuilder
modelBuilder)
{
            modelBuilder.Entity<OrderTransaction>()
                 .HasOne<Order>(sc => sc.Order)
                 .WithMany(s => s.Transactions)
                 .HasForeignKey(sc => sc.OrderID);
            modelBuilder.Entity<OrderTransaction>()
                 .HasOne<Transaction>(sc => sc.Transaction)
                 .WithMany(s => s.Orders)
                 .HasForeignKey(sc => sc.TransactionID);
            modelBuilder.Entity<UserAccount>()
                 .HasMany(m => m.Orders)
                 .WithOne(t => t.UserAccount);
            modelBuilder.Entity<Order>()
                 .HasOne(m => m.UserAccount)
                 .WithMany(t => t.Orders);
        }
```

Ispis 9: Primjer koda metode OnModelCreating u ExchangeContext.cs

Nakon toga se može postaviti `DbSet` za svaki pojedinačni model (Ispis 10):

```
public DbSet<UserAccount> UserAccounts { get; set; }
public DbSet<Order> Orders { get; set; }
public DbSet<Transaction> Transactions { get; set; }
public DbSet<OrderTransaction> OrdersTransactions { get; set; }
public DbSet<UserTransactions> UserTransactions { get; set; }
public DbSet<BalanceHistory> BalanceHistory { get; set; }
public DbSet<ExchangeRates> ExchangeRates { get; set; }
```

Ispis 10: Primjer svojstava ExchangeContext.cs

`DbContext` je potrebno dodati kao servis (registrirati) u `services` unutar Startup.cs datoteke, ali prije toga se mogu definirati kontroleri za stvorene modele te tako imati kompletan set alata za izgradnju poslužitelja.

Svaki definirani model će imati svoju pripadajuću `Controller` datoteku iako je većina stvarnih pristupnih točaka definirana u zasebnoj datoteci `HomeController.cs`. Skoro sve korisničke pristupne točke se nalaze unutar `HomeController` datoteke, a pristupne točke za admin korisnike su unutar ostalih (model) kontrolera.

Za ubrzano generiranje nekih najčešćih REST HTTP metoda (GET, POST, PUT & DELETE) na već definirane modele, može se iskoristiti funkcionalnost Visual Studio 2019 (preciznije .NET projekta) i Entity Frameworka za jednostavnu izgradnju kontrolera pomoću instalacijskog čarobnjaka. Za to napraviti treba u projektu kliknuti na botun `Add new item`, potom odabrati `Controller` te na kraju izabrati opciju `MVC Controller with views, using Entity Framework` te pratiti čarobnjak. Dodavanje dodatne logike iza stvorenih HTTP metoda mora obaviti korisnik.

Pošto je većina metoda unutar kontrolera definirana na sličan način, samo primjetne razlike će biti spomenute kroz neke primjere. Na primjer, GET ALL pristupna točka unutar `UserAccountsController` i `OrdersController` se razlikuje samo u `DbContext`-u iz kojeg se dohvaćaju podaci (`context.UserAccounts` ili `context.Orders`), pa nema potrebe da se obje verzije detaljnije opisuju. Najosnovnije REST HTTP metode (pristupne točke) će se navesti na primjeru `UserAccountsController`.

- Definicija `UserAccountsController` (Ispis 11):

```
[Route("api/[controller]")]
[ApiController]
[Authorize(Roles = "Admin")]
public class UserAccountsController : ControllerBase
```

Ispis 11: Primjer koda definicije kontrolera (u ovom slučaju UserAccountsController.cs)

Atribut `[Route]` definira url za pristupnu točku tog kontrolera. `[controller]` "wildcard" će poprimiti ime datoteke kontrolera (u ovom slučaju *UserAccounts*) te putanja postaje *`api/UserAccounts`*.

Atribut `[ApiController]` govori da se ova klasa označava `Controller` klasu.

Atribut `[Authorize]` ograničava pristup za samo one korisnike pod navedenim rolama (u ovom primjeru samo Admin rola ima autoriziran pristup pristupnim točkama).

Oznaka `:ControllerBase` govori da trenutna klasa nasljeđuje klasu `ControllerBase` i s time sva njena izložena svojstva i metode.

Svaka klasa `Controller` ima definiranu varijablu `_context` kao i konstruktor u kojem se inicijalizira `_context` na prethodno definirani `DbContext` (Ispis 12).

```
private readonly ExchangeContext _context;

public UserAccountsController(ExchangeContext context)
   {
       _context = context;
   }
```

Ispis 12: Primjer koda kontruktora kontrolera (UserAccountsController)

Nakon toga, korištenje i povezivanje s bazom podataka se može s lakoćom obavljati. Bit će detaljnije objašnjeno na slijedećim primjerima:

- GET ALL UserAccounts pristupna točka (Ispis 13):

```
// GET: api/UserAccounts

[HttpGet]

public async Task<ActionResult<IEnumerable<User>>>
GetUserAccounts()

{

var userAccounts = await _context.UserAccounts.ToListAsync();

return Ok(TransformMultiple(userAccounts));

}
```

Ispis 13: Primjer koda "GET all" metode UserAccountController.cs

Definicija metode nam govori da ta funckija vraća objekt `Task`. `Task` predstavlja asinkronu operaciju koja može vratiti nekakvu vrijednost. Ova definicija je određena razvojnim okvirom .NET Core i zahtjeva da REST metode budu tipa `Task`. Vrijednost koju zatim `Task` vraća je objekt tipa `ActionResult` koji također definira tip vrijednosti koju će

on proslijedit dalje. `ActionResult` je sintaksa također definirana .NET Coreom. Tip vrijednosti koju vraća `ActionResult` ovisi o potrebama metode te je u slučaju ove pristupne točke to tip `IEnumerable<User`. Tip `IEnumerable` obuhvaća sve tipove kolekcija kroz čije podatke se može iterirati (npr. liste, nizovi, ali nije ograničen na njih te obuhvaća i kompleksnije strukture podataka). Tip podataka od kojih se sastoji kolekcija je definiran u ovom slučaju kao `User`, dodatna klasa prethodno definirana unutar `UserAccountsController` (Ispis 14):

```
new public class User
        {
            public Guid ID { get; set; }
            public string Name { get; set; }
            public long AccountBalanceUSD { get; set; }
            public long OrderBalanceUSD { get; set; }
            public long AccountBalanceBTC { get; set; }
            public long OrderBalanceBTC { get; set; }
        };
```

Ispis 14: Primjer pomoćne klase kontrolera UserAccountsController

Ta klasa je definirana kako mi mogli modelirati podatke iz baze prije slanja krajnjem korisniku radi zaštite osjetljivih informacija o korisnicima. Također, treba definirati i dvije pomoćne statične funkcije `TransformAccount` i `TransformMultiple` (Ispis 15) zbog daljnje pretvorbe podataka (i dodatne prilagodbe vrijednosti u realne brojeve).

```
public static Object TransformAccount(UserAccount userAccount)
{
      return new
      {
            Name = userAccount.Name,
            AccountBalanceUSD =
      (float)userAccount.AccountBalanceUSD / 100,
            OrderBalanceUSD =
      (float)userAccount.OrderBalanceUSD / 100,
            AccountBalanceBTC =
      (float)userAccount.AccountBalanceBTC / 100000000,
            OrderBalanceBTC =
      (float)userAccount.OrderBalanceBTC / 100000000,
            isAdmin = userAccount.Role == "Admin"
      };
}
public static IEnumerable<Object>
TransformMultiple(IEnumerable<UserAccount> accounts)
{
      return accounts.Select(UserAccount =>
TransformAccount(UserAccount));
}
```

Ispis 15: Primjer koda pomoćnih metoda UserAccountsController.cs

Upotreba metode `TransformMultiple` unutar `getUserAccounts` (Ispis 16):

```
{
var userAccounts = await _context.UserAccounts.ToListAsync();
return Ok(TransformMultiple(userAccounts));
}
```

Ispis 16: Primjer koda za dohvaćanje korisničkih računa iz baze

Poziv `await _context.UserAccounts.ToListAsync()` je prvi i zapravo jedini poziv unutar funckije. On govori da unutar prethodno definiranog konteksta, dohvati njegov `UserAccounts` set (sve podatke iz baze vezano za tablicu `UserAccounts`) i vrati ga kao listu. `ToListAsync()`i `await` su povezani i govore .NET Coreu da čeka sa svim drugim operacijama nad tim setom (set `UserAccounts`) dok se ova operacije ne izvrši. Nakon toga šalje se lista korisničkih računa u pomoćnu metodu `TransformMultiple` te će se na kraju dobiti lista svih korisničkih računa moduliranih za potrebe krajnjeg korisnika. Povratna vrijednost je HTTP `status code 200` (metoda `Ok`) zajedno s objektom `userAccounts`. Pristup toj metodi, ali i mnogim drugima je dostupan zbog nasljeđivanja klase `ControllerBase` na klasu `UserAccountsController`.

Na sličan način će se napraviti ostale pristupne točke (neke će biti automatski generirane ukoliko je napravljen `MVC Controller with Entity Framework wizard`) samo pozivajući drugačije metode od `_context.UserAccounts` te dodavajući ekstra kôd za provjere i modulacije podataka.

Za `UserAccountsController` definirane REST metode su:

**Komentari na vrhu ispisa predstavljaju primjer URL-a pristupne točke*

- GET *All* (već obrađeno iznad)
- GET *by ID* (Ispis 17)

```
// GET: api/UserAccounts/5
[HttpGet("{id}")]
public async Task<ActionResult<UserAccount>>
GetUserAccount(Guid id)
  {
    var userAccount = await
_context.UserAccounts.FindAsync(id);
     if (userAccount == null)
     {
         return NotFound("No user found with ID = " + id);
     }
     return Ok(TransformAccount(userAccount));
  }
```

Ispis 17: Primjer koda metode ”GET single”

Metoda `FindAsync` (ili za sinkronu varijantu samo `Find`) pronalazi entitet iz tablice s odgovarajućim atributom ID te vraća `null` ukoliko nije pronađen redak (tada je povratna vrijednost `NotFound` koja je također nasljeđena od `ControllerBase` klase, isto kao i `Ok` metoda). Za ubuduće, sve povratne vrijednosti/metode su izvedene iz `ControllerBase` klase.

- GET *by Name* (Ispis 18)

```
// GET: api/UserAccounts/user=admin
[Route("user={name}")]
[HttpGet]

public async Task<ActionResult<UserAccount>>
GetUserAccountByName(string name)

{

      UserAccount userAccount;

      try

      {

            userAccount = await
      _context.UserAccounts.Where(user => user.Name ==
      name).SingleAsync();

      } catch (Exception e)

      {

            return StatusCode(500, "Failed getting user" +
      "\n" + "ERROR: " + e);

      }

      if (userAccount == null)

      {

            return NotFound("No user found by the name '" +
      name + "'");

      }

      return Ok(TransformAccount(userAccount));

}
```

Ispis 18: Primjer koda ”GET single *by name”*

Metoda `Where` obavlja filtraciju podataka koji zadovoljavaju uvjet prema poslanoj funkciji (engl. *predicate*) (lamda funkcija koja vraća boolean vrijednost provjere za poslanu vrijednost). Ta funkcija će se izvršiti na svakom članu niza te će konačni rezultat biti niz entiteta koji su zadovoljili uvjet.

Metoda `SingleAsync` vraća pronađeni entitet ili `exception` ukoliko je pronađeno više od jednog elementa.

- GET *name uniqueness check* (Ispis 19)

```
// GET: api/UserAccounts/unique/admin
[Route("unique/{name?}")]
[HttpGet]
[AllowAnonymous]
public async Task<ActionResult> CheckUniqueName(string
name = "")
{
      if (name == "") return BadRequest("No username
provided");
      UserAccount userAccount = await
_context.UserAccounts.Where(user => user.Name ==
name).FirstOrDefaultAsync();
      if(userAccount == null)
      {
            return Accepted("Name '" + name + "' is
      unique");
      }
      else return Ok("Name '" + name + "' exists");
}
```

Ispis 19: Primjer koda provjere jedinstvenosti korisničkog imena

Atribut `[AllowAnonymous]` omogućuje pristup svima, van ograničenja postavljenog na cijelu klasu (s `[Authorize(Role="Admin")]` atributom).

Metoda `FirstOrDefault` radi slično kao i metoda `Single` s razlikom da ne vraća `exception` ukoliko je više od jednog rezultata prisutno, već vrati prvi mogući rezultat. Spomenuta GET metoda vraća status kôd 202 ukoliko je korisničko ime jedinstveno, a status kôd 200 ukoliko već postoji korisnik s tim korisničkim imenom.

- PUT *toggle account active status* (Ispis 20)

```
// PUT: api/UserAccounts/togglestatus/5
[Route("togglestatus/{id}")]
[HttpPut]
public async Task<IActionResult> DisableUserAccount(Guid id)
{
      UserAccount userAccount = await
      _context.UserAccounts.FindAsync(id);
if (userAccount == null)
{
      return NotFound("No user found with ID = " + id);
}
      userAccount.ActiveStatus = !userAccount.ActiveStatus;
      _context.Attach(userAccount);
try
{
      await _context.SaveChangesAsync();
}

catch (Exception e)

{

      return StatusCode(500, "Failed saving changes" + "\n" +
"ERROR: " + e);

}

return userAccount.ActiveStatus ? Accepted("Enabled user
account", TransformAccount(userAccount)) : Accepted("Disabled
user account", TransformAccount(userAccount));

}
```

Ispis 20: Primjer koda metode za promjenu statusa korisnika

Metoda `DisableUserAccount` mijenja trenutni status aktivnosti računa s ID-om koji odgovara poslanom `userID` (iz *disabled* u *enabled* i suprotno)

Poziv `_context.Attach` javlja kontekstu (bazi podataka) da je taj entitet promijenjen i spreman za pohranu podataka, a `await _context.SaveChangesAsync()` pohranjuje sve entitete u bazu s obzirom na primjećene promjene (o tom dijelu Entity Framework vodi računa te radi pozive prema bazi samo kada je potrebno)

- POST *new user* (Ispis 21)

```
// POST: api/UserAccounts

[HttpPost]

public async Task<ActionResult<UserAccount>>
PostUserAccount(UserAccount userAccount)

{

      if (userAccount.Name == null || userAccount.Name.Contains("
") || userAccount.Name.Length < 3)

      {

            return BadRequest("Invalid name format (minimum 3
characters without spaces)");

      }

     if (await _context.UserAccounts.AnyAsync(user => user.Name ==
userAccount.Name))

     {

            return BadRequest("Name already taken");

      }

     if (userAccount.Password != null &&
userAccount.Password.Length < 6)

     {

            return BadRequest("Invalid password format (minimum 7
characters)");

      }
```

```
      userAccount.AccountBalanceBTC = 0;
      userAccount.AccountBalanceUSD = 0;
     userAccount.OrderBalanceBTC = 0;
     userAccount.OrderBalanceUSD = 0;
     userAccount.Role = "Admin";
     userAccount.ActiveStatus = true;
      try
      {
            _context.UserAccounts.Add(userAccount);
            await _context.SaveChangesAsync();
      }
     catch (Exception e)
     {
            return StatusCode(500, "Failed saving changes" + "\n" +
"ERROR: " + e);
      }
     var user = new
     {
            userAccount.ID,
            userAccount.Name,
            userAccount.AccountBalanceUSD,
            userAccount.OrderBalanceUSD,
            userAccount.AccountBalanceBTC,
            userAccount.OrderBalanceBTC,
      };
      return Created("Created new user: ", user);
```

Ispis 21: Primjer koda metode za dodavanje novog admin korisika

Ova metoda stvara novog korisnika (u admin ulozi) te vraća objekt novog korisnika zajedno sa status kôdom 201 (*Created*).

`OrdersController`, isto kao i `UserAccountsController` ima skoro identične metode koje se samo pozivaju u drugom kontekstu podataka. Također je ista definicija svojstva `_context` i konstruktor metode među ostalim jednostavnim HTTP metodama, već navedenim u `UserAccountsControlleru` (GET all, GET by ID i tako dalje...). Definicija `OrdersController` prikazana na ispisu 22:

```
[Route("api/[controller]")]
[ApiController]
[Authorize(Roles = "Admin")]
public class OrdersController : ControllerBase
```

Ispis 22: Definicija OrderControllera

`UserAccountsController`, `OrdersController` ima također svoje pripadajuće *Transform** statične metode (Ispis 23) koje modeliraju podatke iz baze za prikaz krajnjem korisniku kao i posebnu metodu `ValidateOrder` koju provjerava da li atribut `orderType` ispravno poslan.

```
public static Object TransformOrder(Order order)
{
      return new {
                  ID = order.ID,
                  UserID = order.UserID,
                  OrderAmount = (double)order.OrderAmount/10000000,
OrderRemainingSize = (double)order.OrderRemainingSize / 100000000,
                  OrderPrice = (double)order.OrderPrice / 100,
                  OrderDate = order.OrderDate,
                  OrderType = order.OrderType,
                  OrderOpenStatus = order.OrderOpenStatus

              };

         }
```

```
public static IEnumerable<Object>
TransformMultiple(IEnumerable<Order> orders)

{
      return orders.Select(order => TransformOrder(order));
}

public static bool ValidateOrder(Order order)

{

  var type = order.OrderType.Split(" ");
  if (type.Length == 3)
  {
       if (order.OrderAmount <= 0 || order.OrderPrice <= 0)
        {
           return false;
         }
      if (type.ElementAt(0) == "Market" || type.ElementAt(0)
== "Limit")
      {
            if (type.ElementAt(1) != "Buy" ||
            type.ElementAt(1) != "Sell") return false;
      }
            if (type.ElementAt(2) != "Order") return false;
  }
      else return false;
      return true;

}
```

Ispis 23: Primjer koda pomoćnih metoda OrdersControllera

`OrdersController` HTTP metode su:

- GET *all open*, GET *all* & GET *by ID* (Ispis 24)

```
// GET: api/Orders
[HttpGet]
[AllowAnonymous]
public async Task<ActionResult<IEnumerable<Order>>>
GetOrders()

{

      var orders = await _context.Orders.Where(order =>
order.OrderOpenStatus == true).ToListAsync();
      return Ok(TransformMultiple(orders));

}



// GET: api/Orders/All
[Route("All")]
[HttpGet]
public async Task<ActionResult<IEnumerable<Order>>>
GetAllOrders()

{

      var orders = await _context.Orders.ToListAsync();
      return Ok(TransformMultiple(orders));

}



// GET: api/Orders/5
[HttpGet("{id}")]
async Task<ActionResult<Order>> GetOrder(Guid id)
{

      var order = await _context.Orders.FindAsync(id);
      if (order == null) return NotFound("No order found with
ID = " + id);
      return Ok(TransformOrder(order));
}
```

Ispis 24: Primjer GET i ”GET *by ID”* OrdersControllera

- GET *by type* & GET *by user and/or type* (Ispis 25)

```
// GET: api/Orders/ByType=(buy or sell)
[Route("ByType={type}")]
[HttpGet]
public async Task<ActionResult<IEnumerable<Order>>>
GetOrdersByType(string type = null)

{

    var orders = await _context.Orders.Where(order =>
order.TypeEquals(type)).ToListAsync();

    return Ok(TransformMultiple(orders));

}

// GET: api/Orders/byUser=5/(buy or sell)
[Route("byUser={id}/{type?}")]
[HttpGet]
public async Task<ActionResult<IEnumerable<Order>>>
GetOrdersByUser(Guid id, string type = null)

{

    var orders = _context.Orders;
    if (type == null)
    {
        var userOrders = await orders.Where(order =>
    order.UserID == id).ToListAsync();
        return Ok(TransformMultiple(userOrders));
    }
    else if (type.ToUpper() == "BUY" || type.ToUpper() ==
    "SELL")
    {
        var userOrders = await orders.Where(order =>
    order.UserID == id).Where(order =>
    order.TypeEquals(type)).ToListAsync();

        return Ok(TransformMultiple(userOrders));

    }

    else return BadRequest("Search parametar 'type' much be
BUY or SELL");

}
```

Ispis 25: Primjer koda složenijih GET metoda (*by type, by user*)

URL parametar `{type?}` je označen kao neobavezan zbog znaka '?' te se može ili ne mora uključiti u parametre zahtjeva. Također, metode `Where` su sada ulančane zbog dodatne potrebe za filtracijom entiteta.

- PUT *order (deactivate order)* (Ispis 26)

```
// PUT: api/Orders/5
[HttpPut("{id}")]
public async Task<ActionResult> PutOrder(Guid id)
{
    if (!OrderExists(id))
    {
        return NotFound();
    }
    Order order = _context.Orders.Find(id);
    if (order.OrderOpenStatus == false ||
order.OrderRemainingSize == 0)
    {
        return BadRequest("Can only cancel ACTIVE
orders");
    }
    else
    {
        UserAccount user =
_context.UserAccounts.Find(order.UserID);
        long balanceChange;
        long balanceLastValue;
        if (order.TypeEquals("Buy"))
        {
            balanceLastValue = user.AccountBalanceUSD;
```

```
            balanceChange = order.OrderRemainingSize *
order.OrderPrice;
            user.AccountBalanceUSD += balanceChange;
            user.OrderBalanceUSD -= balanceChange;

        } else {

            balanceLastValue = user.AccountBalanceBTC;
            balanceChange = order.OrderRemainingSize;
            user.AccountBalanceBTC += balanceChange;
            user.OrderBalanceBTC -= balanceChange;

        }

        BalanceHistory balanceHistory = new
BalanceHistory()

        {

            UserID = user.ID,
     AccountType = order.TypeEquals("Buy") ? "USD" : "BTC",
            ChangeType = "Cancelled order",
            BalanceLastValue = balanceLastValue,
            BalanceChange = balanceChange,
            DateOfChange = DateTime.Now

        };

        order.OrderRemainingSize = 0;
        order.OrderOpenStatus = false;
        _context.Attach(order);
        _context.Attach(user);
        _context.BalanceHistory.Add(balanceHistory);

    }

    try { await _context.SaveChangesAsync();    }

    catch (Exception e)   {

        StatusCode(500, "Failed saving changes" + "\n" +
"ERROR: " + e);

    }  return Accepted("Canceled order",
TransformOrder(order));

}
```

Ispis 26: Primjer koda metode za mijenjanje statusa narudžbe

Ova velika metoda samo uzima narudžbu s ID koji odgovara ID iz zahtjeva te uklanja sva prisutna sredstva i vraća ih vlasniku narudžbe u njegov novčanik. Nakon toga se narudžba proglašava deaktiviranom i rezultati svih promjena se spremaju u bazu.

Osim navedenih prisutna je i statična metoda `CreateOrder` (Ispis 27) koja prima dva parametra (primjer modela nove narudžbe kao i model računa vlasnika) i koristi se prilikom stvaranja nove narudžbe. `OrdersController` nema metodu za stvaranje narudžbe jer se svaka narudžba odnosi na jednog jedinstvenog vlasnika pa je prebačena u `HomeController` gdje se predaje narudžba ovisno o korisniku koji je poslao zahtjev. Više o tome kasnije.

- `CreateOrder` pomoćna metoda:

```
public static Order CreateOrder(Order order, UserAccount
userAccount)
{
    long remainingSize;
    long balanceChange;
    if (order.TypeEquals("Buy"))
    {
        if (userAccount.AccountBalanceUSD <
(double)order.OrderAmount / 100000000 * order.OrderPrice)
        {
            return null; // BadRequest(new { Error =
"Insufficient funds" });
        } else {
            if (order.OrderVariationType("Market"))
            {
                balanceChange = order.OrderPrice;
                remainingSize = order.OrderPrice;
            }
```

```
            else // order.OrderVariationType("Limit") {

                balanceChange = order.OrderAmount *
order.OrderPrice / 100000000;
                userAccount.OrderBalanceUSD +=
balanceChange;
                remainingSize = order.OrderAmount;

    }
    userAccount.AccountBalanceUSD -= balanceChange;

        }

    } else // order.TypeEquals("Sell") {

        if (userAccount.AccountBalanceBTC <
order.OrderAmount) {

            return null; // BadRequest(new { Error =
"Insufficient funds" });
       } else {

            balanceChange = order.OrderAmount;
            userAccount.AccountBalanceBTC -= balanceChange;
            remainingSize = order.OrderAmount;
            if (order.OrderVariationType("Limit")) {

                userAccount.OrderBalanceBTC +=
balanceChange;

            }

        }

    }

    Order newOrder = new Order {

        UserID = userAccount.ID,
        OrderAmount = order.OrderAmount,
        OrderPrice = order.OrderPrice,
        OrderRemainingSize = remainingSize,
        OrderDate = DateTime.Now,
        OrderOpenStatus = true,
        OrderType = order.OrderType

    };

    return newOrder;
```

Ispis 27: Primjer koda pomoćne statične metode *CreateOrder*

Unutar metode prvo slijedi provjera narudžbe - odgovara li iznos narudžbe iznosu korisničkog računa te ukoliko je sve ispravno oduzima se iznos s računa korisnika dok se istovremeno te vrijednosti spremaju u svojstvo `OrdersAmount` objekta `UserAccount` zbog lakšeg praćenja novčanih sredstava. Povratna vrijednost je novi objekt `Order` sa svim usklađenim vrijednostima i svojstvima.

Slične pomoćne statične funkcije ima i `TransactionsController`, kao što su `TransformTransaction` i `CreateTransaction` (Ispis 28).

```
public static (Transaction, OrderTransaction,
OrderTransaction)
        CreateTransaction(Order BuyOrder, Order SellOrder,
long          soldAmount, long soldPrice)
        {
            Transaction transaction = new Transaction
            {
                Date = DateTime.Now,
                TradedBitcoin = soldAmount,
                TradedPrice = soldPrice
            };

            OrderTransaction buyOrderTransaction = new
                 OrderTransaction
            {
                OrderID = BuyOrder.ID,
                Order = BuyOrder,
                TransactionID = transaction.ID,
                Transaction = transaction
            };

            OrderTransaction sellOrderTransaction = new
                 OrderTransaction
            {
                OrderID = SellOrder.ID,
                Order = SellOrder,
                TransactionID = transaction.ID,
                Transaction = transaction
            };

            return (transaction, buyOrderTransaction,
                 sellOrderTransaction);

        }
```

```
public static Object TransformTransaction(Transaction
transaction)
        {
            return new
            {
                ID = transaction.ID,
                TradedBitcoin =
           (double)transaction.TradedBitcoin / 100000000,
                TradedPrice =
           (double)transaction.TradedPrice / 100,
                Date = transaction.Date
            };
        }

public static IEnumerable<Object>
TransformMultiple(IEnumerable<Transaction  transactions)
        {
            return transactions.Select(transaction =>
          TransformTransaction(transaction));
        }
```

Ispis 28: Pomoćne metode TransactionsControllera

`CreateTransaction` prima odgovarajuće narudžbe koje sudjeluju u transakciji kao i količinu razmijenjenih Bitcoina te cijenu prodaje/kupnje po Bitcoinu, a vraća reference na novo napravljenu transakciju i pomoćne `OrderTransaction` parametre potrebne za spremanje podataka u bazu.

Osim pomoćnih funkcija, `TransactionsController` ima i standarne GET metode:

- GET *all*, GET *by ID* & GET *by UserID* (Ispis 29)

```
// GET: api/Transactions
[HttpGet]
[AllowAnonymous]
public async Task<ActionResult<IEnumerable<Transaction>>>
GetTransactions()
{
            var transactions = await
      _context.Transactions.ToListAsync();
            return Ok(TransformMultiple(transactions));
}

// GET: api/Transactions/5
[HttpGet("{id}")]
public async Task<ActionResult<Transaction>> GetTransaction(Guid
id)
{
      var transaction = await
      _context.Transactions.FindAsync(id);
      if (transaction == null)
      {
            return NotFound();
      }
     return Ok(TransformTransaction(transaction));
}

// GET: api/Transactions/byUser=5
[Route("byUser={id}")]
[HttpGet]
public async Task<ActionResult<IEnumerable<Transaction>>>
GetTransactionByUser(Guid id)
{
      var transactions = await _context.UserTransactions
                  .Where(transaction => transaction.UserID == id)
                  .Include(transaction => transaction.Transaction)
                  .Select(transaction => transaction.Transaction)
                  .ToListAsync();
      return Ok(TransformMultiple(transactions));
}
```

Ispis 29: Primjer osnovnih GET metoda TransactionsControllera

- GET *latest transactions' stats* (Ispis 30)

```
// GET: api/Transactions/stats/1
[Route("stats/{interval}")] // interval in days
[HttpGet]
[AllowAnonymous]

public async Task<ActionResult<IEnumerable<Transaction>>>
GetTransactionsStats(long interval)
{
      var transactions = await
_context.Transactions.Where(transaction => transaction.Date >
DateTime.Today.AddDays(-interval)).ToListAsync();
      if (!transactions.Any()) return Ok(new {
            last = (double)0,
            high = (double)0,
            low = (double)0,
            volume = (double)0
            });
      var stats = new
      {
            last = (double)transactions.Last().TradedPrice / 100,
            high = (double)transactions.OrderByDescending
            (transaction => transaction.TradedPrice)
            .FirstOrDefault().TradedPrice / 100,
            low = (double)transactions.OrderBy(transaction =>
transaction.TradedPrice).FirstOrDefault().TradedPrice / 100,
           volume = transactions.Aggregate(0.0, (acc,
transaction) => acc + (double)transaction.TradedBitcoin /
100000000)
      };

      return Ok(stats);

}
```

Ispis 30: Primjer metode za dohvaćanje statistike o transakcijama u poslanom intervalu

Povratna vrijednost je novi objekt sa svojstvima `last` (zadnja prodana cijena), `high`(est) (najveća prodana cijena), `low`(est) (najmanja prodana cijena) te `volume` (volumen Bitcoina u transakcijama) za poslani interval. Ukoliko nema transakcija u tom intervalu, bit će vraćena prazna lista.

Ostatak značajnih HTTP metoda za bazu podataka je sadržan u klasi `HomeController`, ali prije toga bi trebalo spomenuti i kontroler `OAuth` koji nije vezan za ni jedan model podataka, međutim služi za generiranje i validaciju tokena kao i praćenje već izdanih ili isteklih tokena.

`OAuth` također sadrži `_context` svojstvo kao i konstuktor sličan ostalim kontroler klasama.

- HTTP POST metoda `GenerateToken` (Ispis 31):

```
[HttpPost]

public async Task<ActionResult> GenerateToken(UserAccount
unauthorizedUser)
{
      UserAccount user;
      try   {
            user = await _context.UserAccounts.Where(user =>
user.Name == unauthorizedUser.Name).SingleAsync();
            if (user.Password != unauthorizedUser.Password)
throw new Exception();
            if (!user.ActiveStatus) return StatusCode(403,
"Account is disabled");
      }
      catch (Exception){
            return BadRequest("Incorrect username or
password");
      }

      var mySecret =
Startup.Configuration["JwtToken:SecretKey"];

      var mySecurityKey = new
SymmetricSecurityKey(Encoding.ASCII.GetBytes(mySecret));

      var myIssuer = Startup.Configuration["JwtToken:Issuer"];

      var myAudience =
Startup.Configuration["JwtToken:Issuer"];

      var claims = new[]{
            new Claim(JwtRegisteredClaimNames.Sub, user.Name),
            new Claim(JwtRegisteredClaimNames.Jti,
            user.ID.ToString()),
            new Claim(ClaimTypes.Role, user.Role)
      };
```

```
      var signingCredentials = new
      SigningCredentials(mySecurityKey,
      SecurityAlgorithms.HmacSha256);
      var token = new JwtSecurityToken(
            myIssuer,
            myAudience,
            claims,
            notBefore: DateTime.Now,
            expires: DateTime.Now.AddMinutes(60),
            signingCredentials);
      var tokenJson = new
      JwtSecurityTokenHandler().WriteToken(token);

      var account = new
      {
        user.ID,
        user.Name,
        AccountBalanceUSD = (float)user.AccountBalanceUSD/100,
        OrderBalanceUSD = (float)user.OrderBalanceUSD / 100,
        AccountBalanceBTC = (float)user.AccountBalanceBTC /
         100000000,
        OrderBalanceBTC = (float)user.OrderBalanceBTC /
         100000000,
        isAdmin = user.Role == "Admin"

      };

      return Ok(new { token = tokenJson, user = account});

}
```

Ispis 31: Primjer koda metode za generiranje autorizacijskog tokena

Parametar za metodu `GenerateToken` je objekt `UserAccount` s obaveznim svojstvima `username` i `password` koji se zatim provjeravaju u bazi podataka. Za izradu autorizacijskog tokena (`JwtSecurityToken` objekt) potrebni su dodatni parametri `signingCredentials` *(*za koje nam trebaju parametari `Secret` i simetrični sigurnosni ključ (engl. *Symmetric security key*) `SecurityKey`*,* `Issuer`*,* `Audience` te niz `Claim`-ova koji sadrže informacije za zapis u tokenu (`userName`, `userID` & `userRole`). Također treba dodati vrijeme stvaranja tokena i vrijeme isticanja valjanosti (postavljeno na 60 minuta). Ukoliko su poslani korisnik i njegove informacije ispravne onda treba samo pozvati metodu

`WriteToken` od klase `JwtSecurityTokenHandler` koja će generirati traženi token. Povratna vrijednost je generirani token kao i objekt s informacijama za odgovarajućeg korisnika (korisničko ime, stanje na računu i tako dalje). Tu se nalazi i pomoćna funkcija `ValidateCurrentToken` (Ispis 32) koja vraća `UserID` iz dekôdiranog tokena ukoliko je ispravan, a `null` ukoliko je neispravan. Metoda se koristi unutar klase `HomeController` za provjeru identiteta korisnika koji šalje zahtjev.

```
public static string ValidateCurrentToken(string token)
{
      var mySecret = Startup.Configuration["JwtToken:SecretKey"];
      var mySecurityKey = new
SymmetricSecurityKey(Encoding.ASCII.GetBytes(mySecret));    var
myIssuer = Startup.Configuration["JwtToken:Issuer"];
      var myAudience = Startup.Configuration["JwtToken:Issuer"];
      if (token.StartsWith("Bearer ")) {
            string bearerToken = token.Split(" ").ElementAt(1);
            string userID = null;
            var tokenHandler = new JwtSecurityTokenHandler();
            try  {
                  var validatedToken =
                  tokenHandler.ValidateToken(bearerToken, new
                  TokenValidationParameters
                  {
                        ValidateIssuerSigningKey = true,
                        ValidateIssuer = true,
                        ValidateAudience = true,
                        ValidIssuer = myIssuer,
                        ValidAudience = myAudience,
                        IssuerSigningKey = mySecurityKey
                  }, out SecurityToken validated);
                  userID = validatedToken.Claims.First(claim =>
                  claim.Type == "jti").Value;
            } catch    {
                  return userID;
            }

            return userID;
      } else return null;
}
```

Ispis 32: Primjer pomoćne funkcije za validaciju tokena

Obrađivanje tokena se odvija preko istog `JwtSecurityTokenHandlera` koji je generirao token, samo ovog puta se na novu instancu `handlera` poziva metoda

`ValidateToken` koja prima token kao i dodatne opcije za validaciju te vraća dekodirani token ukoliko je ispravan. Bacit će iznimku (engl. *exception*) ako je validacija neispravna.

Kompletna funkcija služi samo za metodu `ValidateToken` koja vraća informacije o korisniku iz tokena te se u teoriji mogla i napisati unutar samog tijela `ValidateToken` metode (Ispis 33):

```
// POST: api/OAuth/validate
[Route("validate")]
[HttpPost]
public async Task<ActionResult> ValidateToken([FromBody] string
token)
{
      if (token == null) return BadRequest("No token provided");
      var userID = ValidateCurrentToken("Bearer " + token);
      if (userID == null) return BadRequest("Auth error");
      var user = await
_context.UserAccounts.FindAsync(Guid.Parse(userID));
      if (user == null) return BadRequest("Token error, no user
ID found matching token ID");
      if (!user.ActiveStatus) return StatusCode(403, "Account was
disabled");
      var account = new {
        user.ID,
        user.Name,
        AccountBalanceUSD = (float) user.AccountBalanceUSD / 100,
        OrderBalanceUSD = (float) user.OrderBalanceUSD / 100,
        AccountBalanceBTC = (float) user.AccountBalanceBTC /
            100000000,
        OrderBalanceBTC = (float) user.OrderBalanceBTC /
            100000000,
        isAdmin = user.Role == "Admin"
      };

      return Ok(new { user = account });

}
```

Ispis 33: Primjer koda metode pristupne točke za validaciju tokena

Povratna vrijednost je korisnički račun s ID-om čijem pripada provjereni token.

Klasa `HomeController` (definicija u ispisu 34) obuhvaća sve pristupne točke koje bi krajnjem korisniku bile korisne za uspješno korištenje aplikacijom.

```
[Route("api/[controller]")]
[ApiController]
[Authorize(Roles = "Admin, User")]
public class HomeController : ControllerBase
```

Ispis 34: Primjer definicije HomeControllera

Tu se nalaze metode za dohvaćanje informacija o korisniku, dohvaćanje korisnikovih narudžbi i transakcija, stvaranje novog korisnika (u ulozi korisnika), stvaranje novih narudžbi, deaktivaciju postojećih narudžbi, promjenu šifre korisničkog računa, dodavanje novih novčanih sredstava na korisnički račun te deaktivacija računa (ne radi kao aktivacija, za to treba koristiti već navedenu metodu u `UserAccountsController` odnosno s pristupom admin ulozi).

Bitno je naglasiti da su ove metode dostupne samo autoriziranim korisnicima (izuzevši metodu za stvaranje novih korisnika koja je dostupna svima), te će identitet korisnika biti dohvaćen iz tokena prisutnog u zahtjevu. Za razliku od ostalih klasa *controller,* kod kojih je samo korisnik s admin ulogom imao pristup i vezali su se samo za pripadajući model (na primjer `OrdersController` je obavljao funkcije samo s `Order` entitetima), kod kontrolera `HomeController` je situacija drugačija jer se koriste metode za dohvaćanje informacija svih modela koji se odnose na korisnika.

Kao i svi predhodni kontroleri, `HomeController` također sadrži `_context` svojstvo i svoj konstruktor za dohvaćanje konteksta baze. Osim toga, a različito ostalim obrađenim metodama, `HomeController` ima posebnu već spomenutu provjeru identiteta korisnika (Ispis 35) na način da se korisnički ID iz tokena provjeri prema bazi. Kada se ustvrdi identitet korisnika zahtjeva, ostale funkcionalnosti se obavljaju nad korisničkim podacima (npr. korisnik može stvoriti novu narudžbu samo u svoje ime ili deaktivirati samo svoje narudžbe).

```
UserAccount userAccount;
var token = Request.Headers["Authorization"];
var tokenID = OAuth.ValidateCurrentToken(token);
if (tokenID == null)
{
    return BadRequest("Authorization error");
}
else {
 // GLAVNI DIO METODE
}
```

Ispis 35: Primjer koda provjere identiteta korisnika

Ostale metode kontrolera `HomeController` su:

- GET *account (info)*
    - Metoda identična kao i kod `UserAccountsControllera`, dohvaća se korisnik s identifikacijskim brojem iz tokena
- GET *user's orders*
    - Kao i s prethodnom metodom i slično kao u `OrdersControlleru` ovdje se dohvaćaju sve narudžbe koje pripadaju korisniku s ID iz tokena. Također je moguće poslati dodatni, izborni (engl. *optional*) url parametar koji određuje koji tip (*buy* ili *sell*) narudžbi će se vratiti u odgovoru.
- GET *user's transactions* (Ispis 36)

```
var userID = Guid.Parse(tokenID);
var transactions = await _context.Orders
      .Where(order => order.UserID == userID)
     .Include(order => order.Transactions)
     .ThenInclude(transaction => transaction.Transaction)
     .Select(o => new {
           OrderID = o.ID,
           Type = o.OrderType,
           Transactions = o.Transactions
           .Select(t => t.Transaction)
     }).Where(x => x.Transactions.Count() > 0).ToListAsync();

return Ok(transactions);
```

Ispis 36: Primjer metode za dohvaćanje korisnikovih transakcija

- Kod dohvaćanja korisnikovih transakcija potrebno je iskoristiti malo kompliciraniju logiku za izvući informaciju koje transakcije pripadaju kojem korisniku (jer tablice `UserAccount` i `Transaction` nisu direktno spojene već imaju indirektnu vezu preko tablice `Order` i međutablice `OrderTransaction`). Da bi se postigla veza među entitetima, treba se iskoristiti funkcionalnosti virtualnih svojstava koja su spomenuta kod opisa modela na početku poglavlja. Za to se koristi metoda `Include` od `Orders` konteksta koja govori Entity Frameworku da uključi određena polja u ispis rezultata iz baze. To znači da će se s prvim pozivom `Include` dobiti `Transactions` kolekcija za svaki entitet `Order` te onda s pozivom `ThenInclude` uključiti polje `Transaction` iz kolekcije `Transactions`. Naknadno, potrebno je pozvati `Select()` metodu koja će odabrati samo bitna svojstva elemenata vraćene kolekcije tako da će se dobiti niz elemenata od kojih je svaki element objekt sa svojstvima `OrderID`, `OrderType` i `Transactions` poljem koje sadrži sve transakcije od te narudžbe.

- POST *new user*
  - Ova metoda je identična metodi za stvaranje novog korisnika iz `UserAccountsController` s jedinom razlikom da dopušta pristup svima, a ne samo autoriziranim korisnicima te stvara novi račun u korisničkoj ulozi (dok `UserAccountsController` metodi može pristupiti samo admin korisnik te ta metoda stvara novog korisnika isključivo u Admin ulozi).
- POST new order (Ispis 37)

```
var userAccount = await
_context.UserAccounts.FindAsync(order.UserID);
Order orderConversion = new Order();
orderConversion.UserID = order.UserID;
orderConversion.OrderAmount = (long)(order.OrderAmount *
100000000);
orderConversion.OrderPrice = (long)(order.OrderPrice * 100);
orderConversion.OrderType = order.OrderType;
bool validation =
OrdersController.ValidateOrder(orderConversion);


if (userAccount == null || !validation) {
    return BadRequest("Order object must have active user's
'UserID', 'OrderAmount', 'OrderPrice', 'OrderType'" +
validation);
}
Order newOrder;
newOrder = OrdersController.CreateOrder(orderConversion,
userAccount);
if (newOrder == null) {
    return BadRequest("Insufficient funds");
}
_context.UserAccounts.Attach(userAccount);
_context.Orders.Add(newOrder);
try {
    await _context.SaveChangesAsync();
} catch (Exception e) {
    return StatusCode(500, "Failed saving changes" + "\n" +
"ERROR: " + e);
}
if (orderConversion.OrderVariationType("Market")) {
    return await MatchMarketOrder(newOrder);
}
else return await MatchLimitOrder(newOrder);
```

Ispis 37: Primjer metode za stvaranje nove korisničke narudžbe

- Kod stvaranja nove narudžbe, određena količina novčanih sredstava se privremeno ukloni iz korisničkog računa kako bi bila dostupna kod spajanja narudžbi i stvaranja transakcija. Komplikacije kod novih narudžbi nastaju kada je te narudžbe potrebno spojiti s drugim narudžbama u bazi kako bi se izvršile transakcije. Tu se koriste dvije posebne statične funkcije unutar klase

`HomeController`, a to su `MatchMarketOrder` i `MatchLimitOrder` koje se pozivaju ovisno o tipu poslane narudžbe. One će biti detaljno objašnjene nakon razrade ostalih HTTP metoda `HomeControllera`.

- PUT *order (cancel)* – deaktivacija narudžbe
  - Otkazivanje narudžbe radi na identičan način kako je u `OrderControlleru` objašnjeno, jedina razlika je što se ovdje provjerava da li je osoba koja šalje zahtjev ujedno i ona koja je stvorila narudžbu. Samo vlasnik narudžbe je može deaktivirati. Admin korisnik može ugasiti svaku narudžbu, ali to mora napraviti preko druge pristupne točke (iz `OrdersControllera`).
- PUT *account* – promjena lozinke (Ispis 38)

```
var userID = Guid.Parse(tokenID);
UserAccount user = await
_context.UserAccounts.FindAsync(userID);
if (password.Length < 6)
{
    return BadRequest("Invalid password format (minimum 7
characters)");
}
if (user.Password == password)
{
    return BadRequest("New password is equal to the old one");
}
user.Password = password;
_context.UserAccounts.Attach(user);
try
{
    await _context.SaveChangesAsync();
}
catch (Exception e)
{
    return StatusCode(500, "Failed saving changes" + "\n" +
"ERROR: " + e);
}
return Accepted("Updated password",
UserAccountsController.TransformAccount(user));
```

Ispis 38: Primjer metode za promjenu korisničke lozinke

- PUT *account* – dodavanje iznosa na račun (Ispis 39)

```
int multiplier = wallet.ToUpper() == "USD" ? 100 : 100000000;
long amount = (long)newDeposit * multiplier;
var userID = Guid.Parse(tokenID);
UserAccount user = await
_context.UserAccounts.FindAsync(userID);
if (wallet.ToUpper() == "USD") user.AccountBalanceUSD =
user.AccountBalanceUSD + amount;
else user.AccountBalanceBTC = user.AccountBalanceBTC + amount;
_context.UserAccounts.Attach(user);
try
{
    await _context.SaveChangesAsync();
}
catch (Exception e)
{
    return StatusCode(500, "Failed saving changes" + "\n" +
"ERROR: " + e);
}
return Accepted("Updated account ballance", new {
    AccountBalanceUSD = (double)user.AccountBalanceUSD / 100,
    AccountBalanceBTC = (double)user.AccountBalanceBTC /
     100000000 });
```

Ispis 39: Primjer metode za dodavanje sredstava na korisnički računa

- PUT *account* – deaktivacija računa (Ispis 40)

```
var userID = Guid.Parse(tokenID);
UserAccount user = await
_context.UserAccounts.FindAsync(userID);
user.ActiveStatus = !user.ActiveStatus;
_context.UserAccounts.Attach(user);
try {
    await _context.SaveChangesAsync();
} catch (Exception e) {
    return StatusCode(500, "Failed saving changes" + "\n" +
"ERROR: " + e);
}
return user.ActiveStatus ? Accepted("Enabled account" +
user.ID.ToString()) : Accepted("Disabled account" +
user.ID.ToString());
```

Ispis 40: Primjer metode za deaktiviranje korisničkog računa

Sada se mogu obraditi najveće i najvažnije metode bitne za sveukupnu funkcionalnost poslužiteljskog dijela aplikacije. Metode `MatchMarketOrder` (Ispis 41) i

`MatchLimitOrder` (Ispis 42) koje traže odgovarajuće narudžbe za spojiti s originalnom narudžbom te glavna pomoćna funkcija `MatchOrders` (Ispis 43) koja zapravo odrađuje logiku spajanja dvije narudžbe.

- MatchMarketOrder

```
string exceptionFlag = null;
string orderType = order.TypeEquals("Buy") ? "Limit Sell
Order" : "Limit Buy Order";
List<Order> lowestOrders;
try {
    lowestOrders = _context.Orders
      .Where(order => order.OrderOpenStatus == true)
      .Where(order => order.OrderType == orderType)
      .OrderBy(order => order.OrderPrice)
      .ThenBy(order => order.OrderDate).ToList();
    if (lowestOrders.Count == 0) {
        return await CheckMarketOrder(order, 404, "Unable to
process order due to insufficient amount of orders with
different order type");
    }
}
catch (Exception e){
    var errorMessage = "Failed getting matching orders for "
+ order + "\n" + "ERROR: " + e;
    return await CheckMarketOrder(order, 500, errorMessage);
```

Ispis 41: Primjer pomoćne metode za spajanje korisnikove *Market* narudžbe

- MatchLimitOrder

```
string exceptionFlag = null;
string orderType = order.TypeEquals("Buy") ? "Limit Sell
Order" : "Limit Buy Order";
List<Order> matchingOrders;
try
{
    matchingOrders = _context.Orders
      .Where(order => order.OrderOpenStatus == true)
      .Where(order => order.OrderType == orderType)
      .Where(o => order.OrderType == "Limit Buy Order" ?
order.OrderPrice >= o.OrderPrice : o.OrderPrice >=
order.OrderPrice)
      .OrderBy(o => order.OrderType == "Limit Buy Order" ?
o.OrderPrice : -o.OrderPrice)
      .ThenBy(order => order.OrderDate).ToList();
    if (matchingOrders.Count == 0) return Created("Placed
Limit Order without matching", order.ID);
}
catch (Exception e){
    var errorMessage = "Failed getting matching orders for "
+ order + "\n" + "ERROR: " + e;
    return StatusCode(500, errorMessage);
}
exceptionFlag = MatchOrders(order, matchingOrders);
```

Ispis 42: Primjer pomoćne metode za spajanje korisnikove *Limit* narudžbe

Ove metode skupljanju liste narudžbi (iz baze podataka) koje zadovoljavaju uvjete spajanja s originalno poslanom narudžbom (gdje je uvjet prikladna cijena). Razlika između dvije funkcije je u uvjetu dohvaćanja liste drugih narudžbi gdje zapravo `MatchLimitOrder` zahtjeva detaljniju filtraciju od `MatchMarketOrder`. Također, `MatchMarketOrder` prije povrata rezultata dodatno provjerava stanje i obrađuje poslanu market narudžbu s `CheckMarketOrder` metodom. Dohvaćena lista, ukoliko postoji, se onda zajedno s originalnom narudžbom šalje u glavnu funkciju – `MatchOrders` koja zatim radi logiku spajanja pojedinačnih narudžbi s poslanom listom i obrađuje transakcije među korisnicima.

- MatchOrder

```
foreach (var matchedOrder in matchingOrders)
{
    if (order.OrderOpenStatus == false) break;
    matchedUser =
_context.UserAccounts.Find(matchedOrder.UserID);
    long soldAmount, soldPrice, orderBalance;
    if (order.TypeEquals("Buy"))
    {
        var maximumOrderPayout =
matchedOrder.OrderRemainingSize * matchedOrder.OrderPrice /
100000000;
        if (order.OrderVariationType("Limit"))
        {
            soldAmount = order.OrderRemainingSize <
matchedOrder.OrderRemainingSize ? order.OrderRemainingSize :
matchedOrder.OrderRemainingSize;
            soldPrice = matchedOrder.OrderPrice;
            order.OrderRemainingSize -= soldAmount;
            order.OrderOpenStatus = order.OrderRemainingSize !=
0;
            matchedOrder.OrderRemainingSize -= soldAmount;
            matchedOrder.OrderOpenStatus =
matchedOrder.OrderRemainingSize != 0;
            orderBalance = (long)((double)soldAmount /
100000000  * soldPrice);
            orderSurplus -= orderBalance;
        }
        else // order.OrderVariationType("Market")
        {
            soldAmount = order.OrderRemainingSize <
maximumOrderPayout ? (long)((order.OrderRemainingSize /
(float)matchedOrder.OrderPrice) * 100000000) :
matchedOrder.OrderRemainingSize;
            soldPrice = matchedOrder.OrderPrice;
            order.OrderRemainingSize = order.OrderRemainingSize
> maximumOrderPayout ? order.OrderRemainingSize -
maximumOrderPayout : 0;
            order.OrderOpenStatus = order.OrderRemainingSize !=
0;
            matchedOrder.OrderRemainingSize -= soldAmount;
            matchedOrder.OrderOpenStatus =
matchedOrder.OrderRemainingSize != 0;
            orderBalance = 0;
            orderSurplus = 0;
        }
```

```
        user.AccountBalanceBTC += soldAmount;
        user.OrderBalanceUSD -= orderBalance;
        if(order.OrderRemainingSize == 0)
        {
            user.AccountBalanceUSD += orderSurplus;
            user.OrderBalanceUSD -= orderSurplus;
        }
        matchedUser.AccountBalanceUSD +=
(long)((double)soldAmount / 100000000 * soldPrice);
        matchedUser.OrderBalanceBTC -= soldAmount;
    }
    else // order.TypeEquals("Sell")
    {
        soldAmount = matchedOrder.OrderRemainingSize <=
order.OrderRemainingSize ? matchedOrder.OrderRemainingSize :
order.OrderRemainingSize;
        soldPrice = matchedOrder.OrderPrice;
        order.OrderRemainingSize -= soldAmount;
        order.OrderOpenStatus = order.OrderRemainingSize != 0 ?
true : false;
        matchedOrder.OrderRemainingSize -= soldAmount;
        matchedOrder.OrderOpenStatus =
matchedOrder.OrderRemainingSize != 0 ? true : false;
        orderBalance = (long)((double)soldAmount / 100000000 *
soldPrice);
        orderSurplus -= orderBalance;
        user.AccountBalanceUSD += orderBalance;
        user.OrderBalanceBTC -= order.OrderType == "Limit Sell
Order" ? soldAmount : 0;
        matchedUser.AccountBalanceBTC += soldAmount;
        matchedUser.OrderBalanceUSD -= orderBalance;
    }

(transaction,buyerOrderTransaction,
sellerOrderTransaction) =
TransactionsController.CreateTransaction(matchedOrder, order,
soldAmount, soldPrice);
```

Ispis 43: Primjer pomoćne metode za obavljanje transakcija među spojenim narudžbama

Ova metoda prolazi po sortiranoj listi narudžbi iz prethodnih `MatchOrder` metoda te ih jednu po jednu spaja s originalnom narudžbom. Unutar petlje koja prolazi po listi narudžbi postoje dva grananja, jedno koje obavlja narudžbe tipa *Buy* te drugo koje obavlja narudžbe tipa *Sell*. Unutar bloka za *Buy* narudžbe postoje dva dodatna grananja ovisno je li originalna

narudžba Market ili Limit vrste jer te dvije vrste narudžbi koriste malo drugačiju logiku za spremanje informacija o narudžbi (`OrderRemainingSize` svojstvo entiteta `Order` odražava preostale dolare za Market narudžbu, a za Limit narudžbu odražava preostale Bitcoine za kupnju). Na kraju, preko statične metode `CreateTransaction` iz `TransactionsControllera` stvara se nova transakcija između dvije narudžbe te se sve povratne vrijednosti (novo stanje korisničkih računa i informacije o novoj transakciji) spremaju u bazu podataka. Nakon toga, kao i kod ostalih metoda gdje su se mijenjali podaci iz baze i ove promjene će biti spremljene u bazu podataka preko metode `SaveChanges`. Ukoliko funkcija `MatchOrders` vrati vrijednost `null`, označava uspješno izvršavanje originalne narudžbe (spajanje sa svim drugim narudžbama ili potrošnja svih sredstava originalne narudžbe).

Nakon obrađene logike spajanja narudžbi, `MatchMarketOrder` i `MatchLimitOrder` metode vraćaju HTTP response, ali kod `MatchMarketOrder` treba se izvršiti još jedna provjera prije nego se zaključi proces. Naime, treba provjeriti da li se originalna Market narudžba izvršila do kraja te ako nije treba vratiti preostala novčana sredstva korisniku. Ta funkcionalnost se obavlja u metodi `CheckMarketOrder` (Ispis 44) koja se koristi kao povratna vrijednost iz `MatchMarketOrder` metode umjesto HTTP odgovora, a `CheckMarketOrder` onda vrati odgovarajući HTTP odgovor nakon obavljene zadaće.

- CheckMarketOrder

```
if (order.OrderOpenStatus && order.OrderRemainingSize > 0)
{
    UserAccount user =
_context.UserAccounts.Find(order.UserID);
    if (order.TypeEquals("Buy"))
    {
        user.AccountBalanceUSD += order.OrderRemainingSize;
    }
    else
    {
        user.AccountBalanceBTC += order.OrderRemainingSize;
    }
    order.OrderRemainingSize = 0;
    order.OrderOpenStatus = false;
    _context.Attach(order);
    _context.Attach(user);
    try
    {
        await _context.SaveChangesAsync();
        if (statusCode == 0) return Created("Completed Market
Order, surplus returned", order.ID);
        return StatusCode(statusCode, actionResult);
    }
    catch (Exception e)
    {
        return StatusCode(500, new {
            message = "Multple errors",
            error1statusCode = 500,
            error1 = "Failed to cancel incomplete order",
            error2statusCode = statusCode,
            error2 = actionResult,
            exception = e.Message
        });
    }
}
else if (statusCode == 0) return Created("Placed & Completed
Market Order", order.ID);
else return StatusCode(statusCode, actionResult);
```

Ispis 44: Primjer pomoćne metode za zatvaranje korisnikove *Market* narudžbe

Kada je `HomeController` završen zajedno s drugim komponentama poslužiteljskog dijela, može ih se spojiti u jednu koherentnu cjelinu da svi dijelovi aplikacije mogu međusobno komunicirati. Konfiguracija svih dijelova poslužiteljskog dijela će se obaviti unutar datoteke `Startup`. (Druga datoteka u root direktoriju projekta je datoteka `Program` čiji sadržaj nije potrebno modificirati). Unutar datoteke `Startup` nalazi se nekoliko bitnih metoda koje treba nadograditi kako bi do sada napisane cjeline mogle funkcionirati, metode `ConfigureServices` (Ispis 45) i `Configure` (Ispis 46).

- ConfigureServices

```
services.AddDbContext<ExchangeContext>(opt =>
opt.UseSqlServer("Data Source=ROYALMAGICIAN;Initial
Catalog=BitcoinExchange;Integrated Security=True"));
services.AddControllers().AddNewtonsoftJson();
services.AddAuthentication("OAuth").AddJwtBearer("OAut
h", config =>
{
    config.TokenValidationParameters = new
TokenValidationParameters
    {
        ValidIssuer =
Configuration["JwtToken:Issuer"],
        ValidAudience =
Configuration["JwtToken:Issuer"],
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configurat
ion["JwtToken:SecretKey"]))
    };
});
services.AddCors(options => {
    options.AddPolicy("AllowMyOrigin",
    builder =>
    {
        builder.AllowAnyOrigin()
                .AllowAnyHeader()
                .AllowAnyMethod();
    });
});
```

Ispis 45: Primjer koda konfiguracije servisa iz Startup.cs

Unutar ove metode se dodaju vlastiti servisi. Prvi servis koji se implementira je `AddDbContext` kojem kao tip konteksta klasu `ExchangeContext`, a kao parametar

prima konfiguraciju u kojoj dobiva ”*connection string*” za server od baze podataka (u obliku sintakse `opt => opt.UseSqlServer(”connection-string”)`).

Drugi servis koji treba dodati je *`AddControllers`* na kojeg se direktno veže *`AddNewtonsoftJson`* te je s time osigurano spajanje kontroler klasa na *`container`* od projekta kao i mogućnost serijalizacije HTTP odgovora od tih kontrolera.

Zadnji bitni servis(i) koje treba dodati je servis(i) za autentikaciju korisnika. Preko `AddAuthentication` metode koja kao parametar prima ime stvorenog autentikacijskog kontrolera (`OAuth`) se može konfigurirati servis za autentikaciju. Zatim treba nadovezati `AddJwtBearer` metodu u kojoj se konfiguriraju parametri validacije tokena (konfiguracijski parametri su zapisani u datoteci `appsettings.json`).

Još jedan servis za naglasiti, a koji neće biti potreban u produkcijskoj verziji produkta, je dodavanje CORS kompatibilnosti zbog mogućnosti pokretanja oba dijela aplikacije na istom računalu. Za konfiguraciju CORS-a treba dodati *policy* (pravilo provjere) koji će se naknadno omogućiti (aktivirati) zajedno sa svim drugim spomenutim servisima.

- Configure

```
app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();

app.UseCors("AllowMyOrigin");

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
```

Ispis 46: Primjer kôda gdje se dodaju konfigurirani servisi na kostur server aplikacije

U ovoj metodi se dodaju svi obrađeni dijelovi aplikacije pod jednu cjelinu.

  - `UseRouting` omogućava da poslužitelj prihvaća naše konfiguracije za `[Route]` atribute u kontrolerima

- `UseAuthentication` & `UseAuthorization` definiraju korištenje autentikacije i autorizacije prema ranije namještenim konfiguracijama
- `UseCors` s parametrom imena već prije definiranog *policy* omogućava korištenje tog skupa pravila.
- `UseEndpoints` zajedno s `MapControllers` definira pristupne točke za poslužitelja te spajanje s konfiguriranim kontrolerima

Nakon ovih postupaka, aplikacija je više manje spremna za pokretanje i korištenje. Zadnja stvar koju treba napraviti je podesiti bazu podataka prema stvorenim modelima (*CODE FIRST APPROACH*). S već navedenim i definiranim modelima te kontekstom za bazu podataka, svi sastojci su prisutni za započeti migraciju podataka na bazu. Za započeti migraciju baze podataka potrebno je utipkati slijedeće CLI komandi u Package Manager konzolu:

- `Enable-Migrations`
- `Add-Migration` *<ime-migracije>* (na primjer "`Add-Migration InitialCreate`")
- `Update-Database`

S te tri naredbe Entity Framework stvara bazu podataka i tablice po uzoru na definirane klase iz direktorija `Models`. Također, `Add-Migration` komanda stvori novi direktorij `Migrations` i novu datoteku unutar njega s imenom migracije iz pokrenute CLI komande. Unutar te datoteke se nalazi metoda `Up` koja definira izgradnju baze podataka i njene tablice. Osim toga, može se i nadopuniti nova metoda `Seed` u kojoj se mogu definirati podaci koji će se popuniti u bazu podataka prilikom pokretanja komande `Update-Database`.

S time su obrađeni svi koraci za stvoriti poslužiteljski dio za web aplikaciju trgovanja Bitcoinom. Preostalo je jedino pokrenuti projekt komandom "`IIS Express`" koja se nalazi u alatnoj traci programa Visual Studio 2019. Poslužiteljski dio aplikacije je sada spreman za rad.

3. Izrada korisničkog sučelja

Za izradu korisničkog sučelja aplikacije korišten je program Visual Studio Code (skraćeno VS Code) koji služi kao napredni uređivač teksta s mogućnosti instalacije raznih nadogradnji (engl. *extensions*) te sadrži integriran terminal za izvršavanje CLI komandi. Također je potrebno instalirati okruženje Node.js koji pruža pristup Node Package Manageru (skraćeno NPM). CLI komanda za instalaciju JavaScript modula i „*third party*“ programa je tada dostupna preko NPM programa (komanda `npm install`).

Glavni modul za izgradnju klijentske aplikacije je biblioteka React za stvaranje dinamičkih web stranica. Prvi korak je preuzeti `create-react-app` s NPM registra (to će se napraviti s globalnom NPM instalacijom kako bi se naknadno mogla stvoriti nova React aplikacija u bilo kojem direktoriju računala).

Pokrenuti u komandnoj liniji:

```
npm install -g create-react-app
```

Skripta će globalno instalirati `create-react-app` program pomoću kojeg se mogu stvoriti nove React aplikacije. Pokretanjem komande `create-react-app <app_name>` instalirat će se najpotrebniji paketi i moduli potrebni za minimalnu funkcionalnost React aplikacije. Unutar novo stvorenog direktorija nalazi se nekoliko važnijih datoteka i poddirektorija:

- `public/`
  - `index.html`
  - `manifest.json`
  - `favicon.ico`
- `src/`
  - `App.js / App.css`
  - `index.js / index.css`

- `package.json`

U `public/` direktoriju se nalazi glavna HTML datoteka te se u njemu, među ostalim stvarima, može definirati naslov web stranice. Datoteka je potpuno otvorena za uređivanje, ali većina elemenata će se definirati u zasebnim JavaScript datotekama kao što su nove React komponente. Također, neke od osnovnih informacija o stranici mogu se definirati u `manifest.json` datoteci gdje se mogu programski zadati neki od parametara za aplikaciju (kao što su `short_name`, `name`, `icons`, `start_url`, `backgroud_color` i tako dalje). `Favicon.ico` datoteka određuje izgled ikone kod favoriziranja web stranice.

Datoteka `package.json` sadrži informacije o aplikaciji za NPM registar. Tu se definiraju (zapravo izmijenjuju zadane vrijednosti) ime aplikacije i potrebni moduli odnosno programske ovisnosti (engl. *dependencies)* aplikacije. Bitno je istaknuti i objekt `scripts` unutar datoteke `package.json` u kojem se nalazi glavna skripta (CLI komanda) za pokretanje aplikacije - `npm run start`. Objekt `dependencies` unutar datoteke ima već nekoliko prisutnih svojstava (ključeva) i njihovih vrijednosti. Ključevi predstavljaju imena modula o kojima ovisi aplikacija (engl. *dependencies*) dok vrijednosti predstavljaju instaliranu odnosno traženu verziju određenog modula. `Create-react-app` instalira nekoliko osnovnih modula od kojih su najvažniji `react`, `react-dom` i `react-scripts`. Da bi se postigao željen izgled te ostvarila očekivana funkcionalnost web stranice, treba dodati slijedeće module (ime & verzija):

- `@material-ui/core`: 4.11.0
- `@material-ui/icons`: 4.9.1
    - Koristi se za generalni dizajn komponenata web stranice te teži uniformnom izgledu svih pregleda na aplikaciji
- `echarts-for-react`: 2.0.16
    - Koristi se za vremenski prikaz kretanja cijena u transakcijama kao i prikaz stanja korisničkog računa u grafičkom obliku
- `react-spinners`
    - Služi za jednostavno korištenje raznih "*loading*" efekata
- `axios`: 0.21.1
    - HTTP klijent za komunikaciju s poslužiteljskim dijelom aplikacije

- `lodash`: 4.17.20
    - Omogućava jednostavno upravljanje nizovima i skupovima podataka unutar koda. Sadrži neke od najkorištenijih metoda za manipulaciju skupovima podataka.
- `redux`: 4.0.5
- `redux-thunk`: 2.3.0
- `react-redux`: 7.2.2
    - `redux` i njemu odgovarajući susjedni moduli omogućavaju spremanje ključnih podataka u globalnu strukturu (memoriju) kojoj mogu pristupiti sve komponente web aplikacije (služi za pamćenje stanja između prikaza i komponenti)
    - unutar `reduxa` se spremaju osnovne informacije o prijavljenom korisniku kao i sve informacije o dohvaćenim otvorenim narudžbama prilikom pokretanja stranice.
- `react-router`: 5.2.0
- `react-router-dom`: 5.2.0
- `react-router-native`: 5.2.0
    - `react-router` i njemu pripadajući moduli služe za upravljanje adresnom trakom u web izborniku
- `react-material-ui-form-validator`: 2.1.1
    - koristi se kao pomoć pri provjeri ispravnosti podataka kod nekih formi unutar aplikacije

Dodavanjem ovih modula i verzija unutar objekta `dependencies` te pokretanjem skripte `npm install` unutar istog direktorija u kojem se nalazi datoteka `package.json`, svi navedeni moduli će se instalirati u novo dodani direktorij `node_modules/`. Kada je proces gotov, projekt je napokon spreman za daljnji razvoj. Za razliku od poslužitelja aplikacije, za korisničko sučelje se neće obrađivati programski kôd u detalje jer nije toliko ovisan za funkcionalnost cjelokupne aplikacije već samo kako će se podaci prikazivati krajnjem korisniku

Glavni direktorij aplikacije, `src/` unaprijed sadrži JavaScript datoteke `App` te `index`. Obje datoteke su iznimno potrebne i trebat će ih izmijeniti da bi se omogućila funkcionalnost nekih predhodno navedenih modula. Unutar datoteke `index.js` se odvija inicijalizacija `axios` i `redux` modula (ovdje se također deklarira glavni element odnosno komponenta

aplikacije te se učitava komponenta `App`, definirana u drugoj datoteci). Datoteka `App.js` služi za pokretanje početnih zahtjeva prema poslužitelju i učitavanje ostalih komponenti unutar same datoteke ovisno o ispisanom URL-u. Primjer kôda App komponente gdje se učitavaju ostale komponente odnosno kontejneri (Ispis 46):

```
<Switch>
  <Route path='/login' component={LoginPage} />
  <Route path='/user' component={UserPage} />
  <Route path='/dashboard' component={Dashboard} />
  <Redirect path='/' to='dashboard' component={Dashboard} />
</Switch>
```

Ispis 46: Pomoćne varijable iz stranih biblioteka koje se koriste u Balance komponenti

Iako web stranica daje uvid korisniku da se kreće po različitim putanjama prilikom navigacije, svi prikazi su iscrtani unutar glavne komponente `App` te se stranica zapravo u niti jednom trenutku ne osvježi.

U duh React logike direktorij `src/` će se podijeliti na nekoliko poddirektorija (iako se u nekim slučajevima kôd ne pridržava toliko strogo konvenciji):

- *`components`* - označava komponente (dijelove) stranice koji se mogu iskoristiti više puta ovisno o potrebi. Za React, komponente mogu biti klasne ili funkcijske (engl. *class or functional components*), ali razlike ne utječu puno na njihovo krajnje korištenje, samo definiciju i deklaraciju.
- *`containers`* - označava veće dijelove stranice koji sadrže više komponenti i nisu namjenjeni višestrukom pozivanju. Obično su to klasne komponente koje sadrže druge, najčešće funkcijske komponente.
- *`CSS`* - direktorij s CSS datotekama
- *`logos`* - direktorij sa SVG datotekama
- *`store`* - direktorij za *redux* kôd

Direktorij `components` ima najviše sadržaja i u njemu se nalaze:

- AccountOptions.js
- Balance.js
- DepositForm.js
- LoginForm.js
- Orders.js
- OrdersForm.js
- OrdersTable.js
- PageHeader.js
- TradingStats.js
- TransactionsChart.js
- TransactionsTable.js
- UserOrdersTable.js
- UserTransactionsTable.js

Te datoteke odnosno komponente primaju podatke od roditeljskih komponenti i stvaraju svoj prikaz na temelju tih podataka ili konteksta u kojem se pozivaju. Za primjer će se obraditi kôd funkcijske komponente `Balance` (Ispis 48).

Komponenta `Balance` koristi više pomoćnih komponenti biblioteke `material-ui` da bi imali jedinstven izgled najkorištenijih HTML komponenti preko čitave aplikacije. Neke od tih komponenti su: `Card` (prilagodljiv `div` element), `Tooltip`, `Icons` i tako dalje. Te komponente se uvode (engl. *import*) na slijedeći način (Ispis 47):

```
import React, { useState } from 'react';
import ReactEcharts from 'echarts-for-react';
import { Grid, Card, CardHeader, Fab, Tooltip } from '@material-
ui/core';

import ExpandLessIcon from '@material-ui/icons/ExpandLess';
import ExpandMoreIcon from '@material-ui/icons/ExpandMore';
import AccountBalanceWalletIcon from '@material-
ui/icons/AccountBalanceWallet';

import CreditCardIcon from '@material-ui/icons/CreditCard';
```

Ispis 47: Pomoćne varijable iz stranih biblioteka koje se koriste u Balance komponenti

Uvode se varijable `React` te `useState` koje služe za obrađivanje JSX koda ili preciznije za varijablu `useState` da omogući funkcionalnost ”stanja komponente“ (engl. *component state*) funkcijske komponente (samo klasne komponente imaju interno stanje komponente). Osim spomenutih varijabli koristi se i varijabla `ReactEcharts` koja služi za crtanje grafa prema dobivenim podacima (u slučaju komponente `Balance` bit će graf oblika `doughnut`, a za vremensku liniju transakcija koristi se `candlestick` graf).

```
<div>
  <Card style={{ backgroundColor: '#96e8bc' }}>
    <CardHeader
      avatar={
        <CreditCardIcon />
      }
      action={ !emptyDollarWallet &&
        <Tooltip title={!showDollarWallet ? 'View chart' : ''}
arrow placement='left-start'>
          <Fab size='small' aria-label='settings' onClick={() =>
toggleDollarWallet(!showDollarWallet)} style={{ backgroundColor:
'#2a9134' }}>
            {showDollarWallet ? <ExpandLessIcon /> :
            <ExpandMoreIcon />}
          </Fab>
        </Tooltip>
      }
      title={"$ " + props.wallet.balanceUSD}
      titleTypographyProps={{ variant: 'h6'}}
      subheader='Dollar Wallet'
      subheaderTypographyProps={{ variant: 'body2' }}
    />
    {showDollarWallet ? <DoughnutChart
      height='40vh'
      color={[
        '#2a9134',
        '#283f3b'
      ]}
      balanceType='USD'
      wallet={props.wallet.balanceUSD}
      orders={props.wallet.ordersUSD}>
    </DoughnutChart> : null}
  </Card>
</div>
```

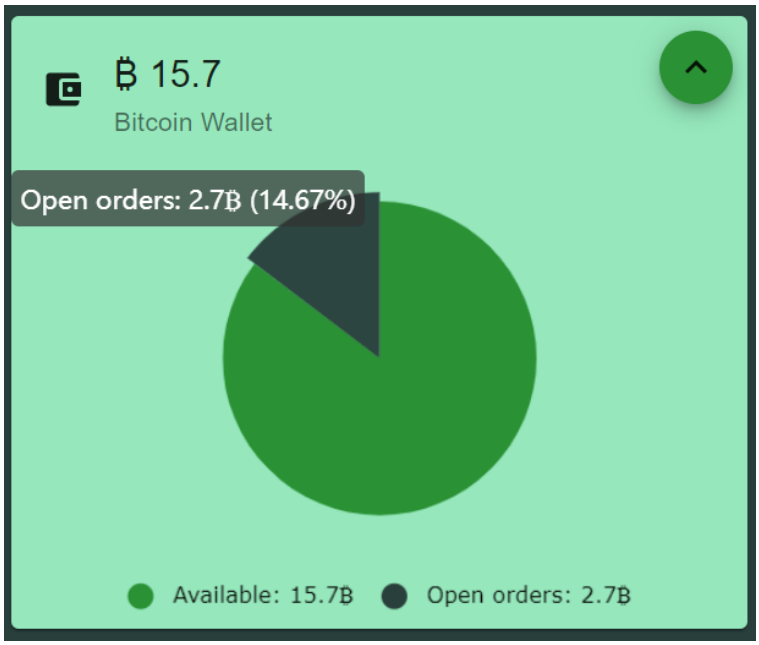
Ispis 48: Primjer JSX koda za prikaz *Bitcoin računa* unutar Balance komponente.

Kôd Balance komponente rezultira u prikazu elemenata na stranici kako je prikazano na Slici 2, odnosno ukoliko se proširi prikaz klikom na ikonu, dobit će se novi prikaz kako je predstavljeno na Slici 3. Iako je u primjeru naveden kôd za prikaz komponente za Bitcoin novčanik, unutar iste datoteke, u istoj povratnoj vrijednosti se nalazi i drugi `div` element s novčanikom Fiat valute kao i neke druge opcije potrebne isključivo za graf. U komponentu `Balance` poslat će se stanje oba računa korisnika te će nam ona vratiti prikaz s obzirom na

primljene podatke. Treba napomenuti da se komponenta `Balance` koristi na više različitih prikaza, na glavnoj stranici kada je korisnik logiran te na stranici korisničkog profila.



Slika 2: Komponenta Balance s dvije minimizirane kartice (Dollar i Bitcoin račun)



Slika 3: Maksimizirana kartica Bitcoin računa. (prikazan je i prijelaz miša na ”open orders” dio grafa)

Direktorij `containers` sadrži tri glavna prikaza aplikacije:

- `Dashboard`
- `LoginPage`
- `UserPage`

Prikaz `Dashboard` se razlikuje ovisno o tome da li je korisnik trenutno prijavljen. Ukoliko je prijavljen tada će se na stranici prikazati dodatna forma za unos novih narudžbi kao i dodatan izbornik kod klika na korisničko ime u navigacijskoj traci. Osim toga, `Dashboard`

prikaz za prijavljenog korisnika sadrži i već spomenutu komponentu `Balance` za pregled stanja računa. `Dashboard` sadrži slijedeće jednostavne komponente:

- `PageHeader` (navigacijska traka)
- `TradingStats`
- `TransactionsChart`
- `Balance`
- `Orders` (a time i `OrdersForm`)
- `OrdersTable`
- `TransactionsTable`

Prikaz `LoginPage` pruža korisnicima formu za prijavu u aplikaciju ili stvaranje novog korisničkog računa. `LoginPage` je zapravo drastično jednostavniji od ostalih kontejnera jer koristi samo dvije komponente definirane u direktoriju `Components`. Komponente na `LoginPage` stranici su forma za prijavu (komponenta `LoginForm`) te navigacijska traka (komponenta `PageHeader`).

Prikaz `UserPage` sadrži sve jednostavnije komponente potrebne za prikaz personaliziranih podataka trenutno prijavljenog korisnika. Među njima su tablice za prikaz korisnikovih narudžbi i transakcija kao i forma za unos novih sredstava na korisnički račun. Ukoliko je prijavljen korisnik s admin pravima, `UserPage` će također prikazivati i komponentu `AdminTools` koja sadrži tablicu o svim trenutnim ostalim korisnicima kao i opcijama za modificiranje svakog pojedinačnog korisnika (aktivacija i deaktivacija računa kao i podizanje ili uklanjanje korisničkih prava). Uz to se može opet pronaći komponenta `Balance` što pridonosi pojmu lako upotrebljivih React komponenti. `UserPage` prikazuje slijedeće komponente:

- `PageHeader`
- `UserTransactionsTable`
- `UserOrdersTable`
- `Balance`
- `DepositForm`
- `AdminTools`

5. Prikaz korisničkog sučelja

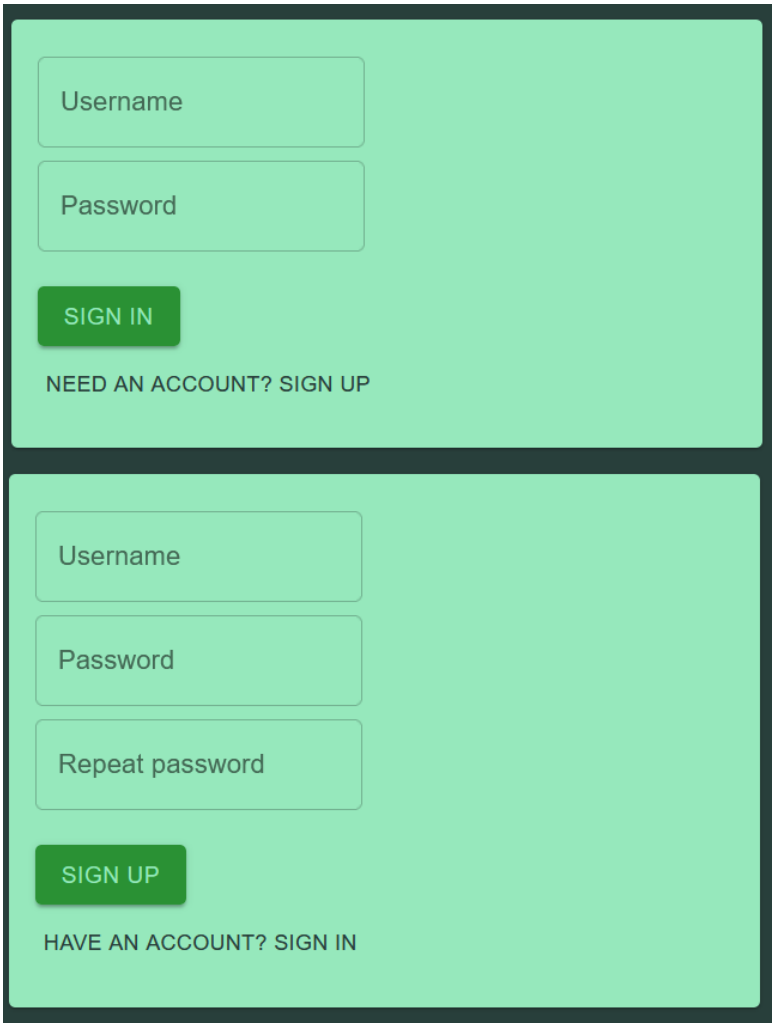
Prvi prikaz kojeg korisnik vidi kada posjeti stranicu je komponenta `Dashboard` bez prijavljenog korisnika (Slika 4). Botun za prijavu (engl. *Sign in*) se nalazi na navigacijskoj traci te je jedina funkcionalnost koja je korisniku trenutno na raspolaganju. Osim prijave, anonimni korisnik može

promatrati stranicu i podatke o narudžbama ili transakcijama bez sudjelovanja u istima.



Slika 4: Glavni prikaz (Dashboard) bez korisnika

Ukoliko se osoba odluči prijaviti, mora to napraviti preko forme za prijavu dostupne preko ikone na navigacijskoj traci. Klikom na tu ikonu se otvara novi prikaz i komponenta `LoginPage` (Slika 5). Na tom prozoru korisnik može direktno unijeti svoje podatke i započeti prijavu ili, ukoliko nema postojeći račun, može napraviti novi preko posebne forme za registraciju novog računa, klikom na botun s tekstom „*No account? Sign up*“.

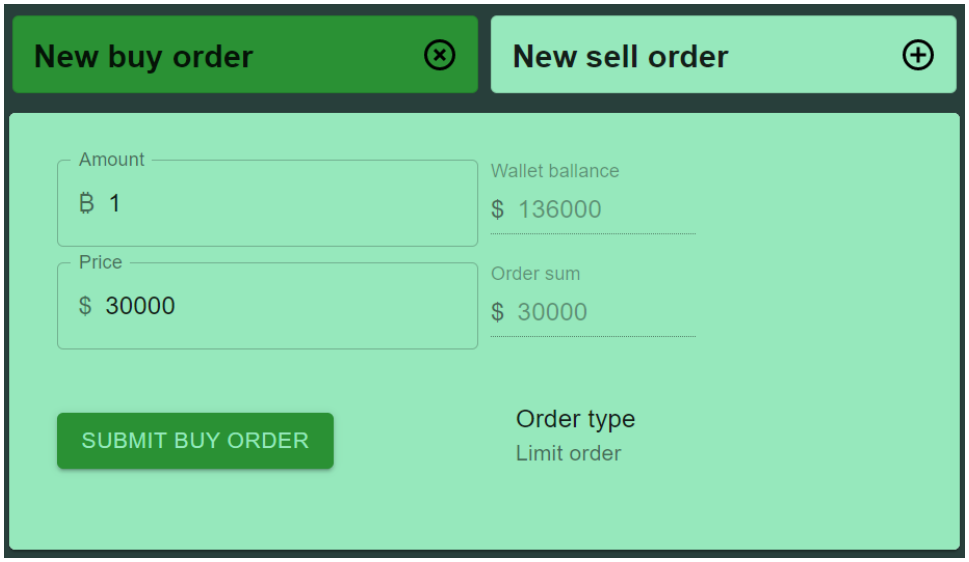


Slika 5: Forma za prijavu / izradu novog korisničkog računa

Nakon što se korisnik prijavio na stranicu, bit će preusmjeren na glavni prikaz, (`Dashboard`) ovog puta s puno više mogućnosti interakcije s aplikacijom (Slika 6). Već spomenuta komponenta `Balance` za prikaz stanja korisničkih računa jedna je od novih komponenti na glavnom prikazu. Druga komponenta je forma za unos novih narudžbi, komponenta `Orders` i njena `OrdersForm`, a nalazi se između grafa transakcija i tablice otvorenih narudžbi. Ta komponenta se dijeli na dva dijela, odnosno dva botuna koji otvaraju odgovarajuću formu – botun s tekstom „*New buy order*“ i njegova forma za podizanje nove narudžbe za kupnju te botun s tekstom „*New sell order*“ te njegova forma za podizanje nove narudžbe za prodaju (Slika 7).

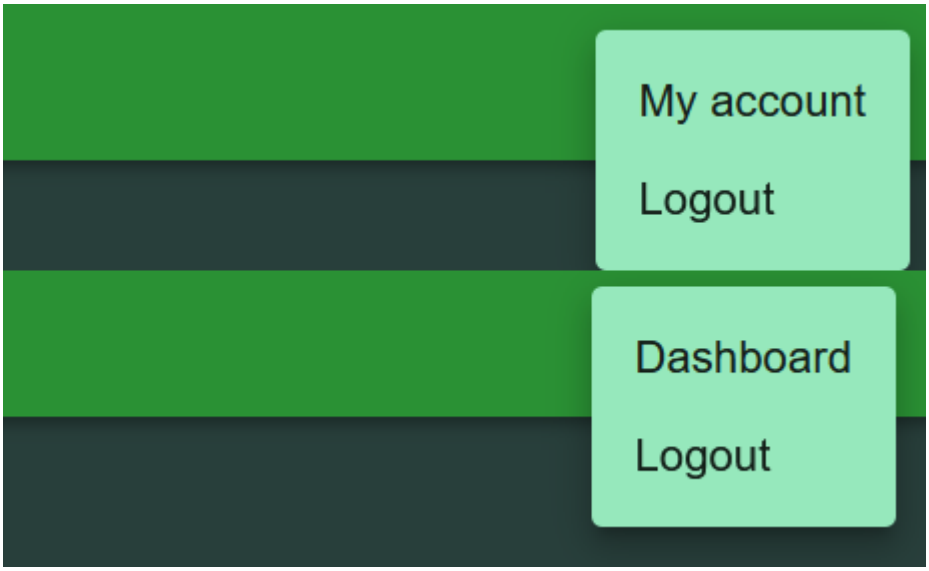


Slika 6: Glavni prikaz s logiranim korisnikom



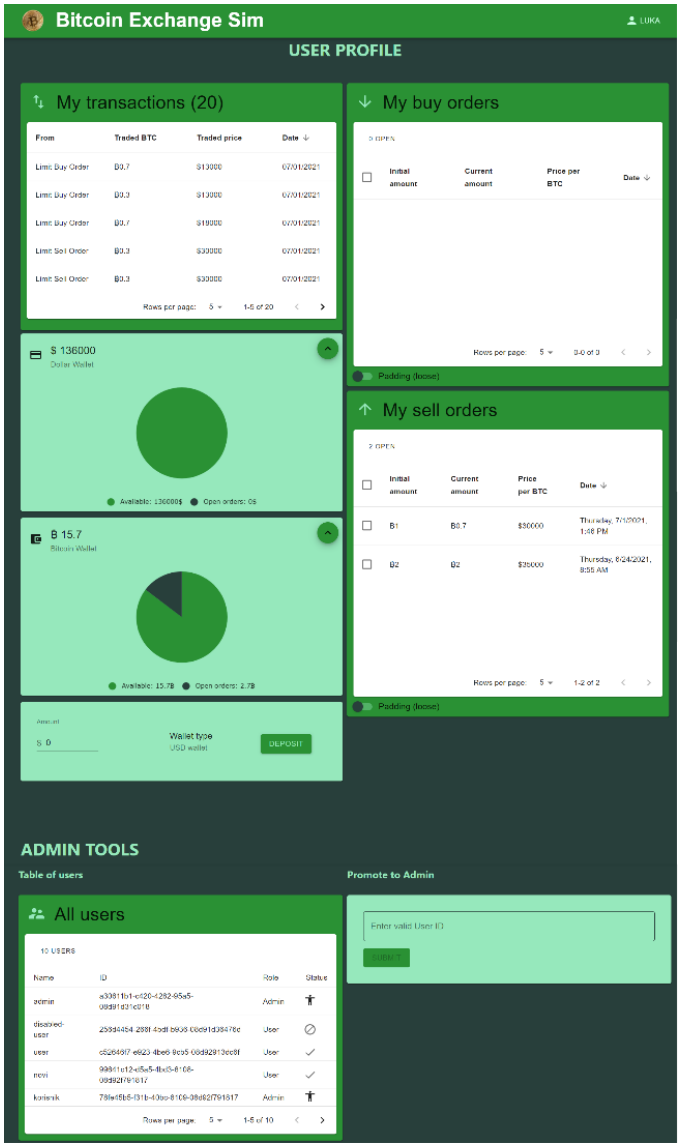
Slika 7: Forma za izradu nove narudžbe (pristupa se klikom na ”New ... order” botune)

Klikom na ikonu profila u navigacijskoj traci pojavljuje se izbornik (Slika 8) s kojim se može pristupiti stranici korisničkog profila (Slika 9), ili glavnoj stranici ukoliko je korisnik već na prikazu profila, kao i obaviti odjavu u bilo kojem trenutku koja potom vodi na `LoginPage` odnosno formu za prijavu.



Slika 8: Primjeri korisničkog izbornika (gore – na glavnom prikazu, dolje – na prikazu korisničkog profila)

Ako korisnik odluči posjetiti svoj prikaz računa na korisničkom sučelju, mora kliknuti na element izbornika s tekstom „*My account*“. Klik će preusmjeriti korisnika na prikaz `UserPage` koji sadrži razne informacije o povijesti trgovanja prijavljenog korisnika, njegove trenutno otvorene narudžbe kao i stanje njegovog računa. Ukoliko je prijavljen korisnik s admin ulogom, također će biti vidljiv i dio s komponentama raznih alata namijenjenih admin korisnicima. Među dodatnim alatima su tablica svih korisnika kao i forma za unaprjeđenje tih korisnika u admin ulogu. Na Slici 9 prikazana je stranica `UserPage` za admin korisnike, ali u usporedbi s običnim korisnicima ona se razlikuje samo u odsječku pod nazivom *Admin tools* koji naravno nisu prisutni normalnim korisnicima.



Slika 9: Primjer korisničkog profila (rola Admin)

6. Zaključak

Bitcoin exchange applikacija uspjela je ostvariti osnovnu funkcionalnost razmjene valuta, odnosno kriptovaluta kao simulaciju stvarnih transakcija. Sve transakcije između korisnika kao i njihova sredstva za trgovanje su ispravno zabilježeni i pohranjeni unutar baze podataka.

Jezik C# kao i razvojno okruženje .NET Core su bili dobar kandidat za izradu ove aplikacije, tj. njegog poslužitelja kao i za njegovu potencijalnu nadogradnju zbog pretežito jednostavne uporabe kao i sveobuhvatne dokumentacije korištene tehnologije. Također, zbog korištenja Visual Studio IDE-a, bilo je trivijalno upotrijebiti i implementirati vanjske servise kao što su provjera JWT tokena ili spajanje s SQL bazom, sve preko terminala ili korisničkog sučelja.

Biblioteka React zajedno s Node.js i jezikom JavaScript pokazali su se kao odličan odabir tehnologija za izradu korisničkog sučelja aplikacije. Zbog prirode jezika JavaScript kao i relativno nove, ali opsežno dokumentirane tehnologije React, uspiješno se izradila web stranica na principu dinamičkih, jednostraničnih web aplikacija.

Također, valja spomenuti da za ozbiljnu produkciju ili daljnji razvoj ove aplikacije bi trebalo implementirati dodatne mjere opreza oko spremanja korisničkih podataka i lozinki, poboljšati sigurnosti promjene lozinki te nadograditi sustav upravljanja korisnicima preko korisničkog sučelja. Te da bi se postigla sinkronizacija pohranjenih podataka s pravim korisnikovim računima, potrebno bi bilo spojiti poslužitelj na dodatan servis koji ce sinkronizirati transakcije i stanje računa između dva sustava.

7. Literatura

[1] C sharp, Wikipedija, https://hr.wikipedia.org/wiki/C_sharp (posjećeno 07.09.2021)

[2] Language Integrated Query, Wikipedia, https://en.wikipedia.org/wiki/Language_Integrated_Query (posjećeno 07.09.2021)

[3] JSX (JavaScript), Wikipedia, https://en.wikipedia.org/wiki/JSX_(JavaScript) (posjećeno 07.09.2021)

[4] Node.js, Wikipedia, https://en.wikipedia.org/wiki/Node.js (posjećeno 07.09.2021)